TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Abstracting real-valued parameters in parameterised boolean equation systems

Laveaux, M.

Award date:
2018

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Formal System Analysis Research Group

Abstracting real-valued parameters in parameterised boolean equation systems

Master Thesis

M. Laveaux

Supervisor:
dr. ir. T.A.C. Willemse

Version 1.0.1

Eindhoven, Friday 9th February, 2018

Abstract

Model checking is the process of verifying whether a property holds in a given model. This can be automated when the model and property are written in a formal language. The minimal common representation language (mCRL2) is a formal language that can be used to specify action models of processes. The modal μ-calculus is a formal language to specify properties.

The mCRL2 tool-set takes these languages as input and is able to verify properties automatically. Internally, this tool-set utilizes so-called *parameterised boolean equation systems* for this purpose. For models of real-timed systems this introduces real-valued parameters in these equation systems. Solving parameterised boolean equation systems is done by instantiating them to a boolean equation system. For real-valued parameters this potentially results in an infinite number of boolean equations.

Alternatively, timed automata are well-studied objects that can be used to model real-timed systems. For timed automata the decidability of model checking questions can be obtained by using a so-called *region* abstraction. In this thesis we show that similar abstraction can be applied to parameterised boolean equation systems such that the instantiated boolean equation system is finite. An alternative abstraction, where regions are combined into so-called *zones*, is defined as well. In some cases this approach works better in practice, but is also more restricted in the type of model checking questions that it can answer.

Both a region and a zone representation have been implemented using the mCRL2 language. Finally, some example parameterised boolean equation systems have been solved using this implementation.

Contents

Contents **2**

1 Introduction **4**
1.1 Related work . . . 5

2 Preliminaries **6**
2.1 Lattice and fixed points . . . 6
2.2 Vector spaces . . . 7
2.3 Data . . . 8

3 Parameterised boolean equation systems **9**
3.1 Parameterised boolean equation systems . . . 9
3.2 Proof graphs . . . 10
3.3 Example . . . 12

4 Timed automata **14**
4.1 Timed automata . . . 14
4.2 Timed transition systems . . . 15
4.3 Semantics of timed automata . . . 16

5 Region equivalence for equation systems **18**
5.1 Translation . . . 21
5.2 Proof of correctness . . . 25

6 Zone abstraction for equation systems **30**
6.1 Translation . . . 31
6.2 Proof of correctness . . . 33
6.2.1 Constructing the original proof graph . . . 37
6.2.2 Constructing the zoned proof graph . . . 39
6.2.3 Combination . . . 41
6.3 Lasso proof graph . . . 41

7 Implementation **43**
7.1 mCRL2 . . . 43
7.2 Regions . . . 44
7.3 Zones . . . 51
7.3.1 Difference bound matrices . . . 51
7.3.2 Implementation . . . 52

8 Example **59**
8.1 Drifting clock 59
8.1.1 Abstracting using regions 60
8.1.2 Abstracting using zones 61
8.1.3 Time comparison 62

9 Conclusion **65**

10 Future work **66**

Bibliography **67**

A Drifting clock example **68**

Chapter 1

Introduction

The mCRL2 toolset is a collection of tools that can be used to verify properties on models specified in its process algebra [6]. This process algebra is called minimal Common Representation Language, abbreviated as mCRL2 [8]. The behaviour of systems with time, for example real-time embedded systems or networking protocols where message passing has some delay, can be specified in this process algebra. However, the set of properties that can be verified on processes with time is limited. Verifying properties is done by applying a number of translations, which are implemented by tools in the tool-set.

The process algebra specification is first translated to a *linear process specification* to aid the later translations. Then a property, written in *modal μ-calculus*, together with this linear process specification is translated to a *parameterised boolean equation system.* The solution for this parameterised boolean equation system contains in which states, or more accurately in which process and which values for its parameters, the property holds. When the property holds for the process with some initial values, which are determined by the model, then we say that the model satisfies the property.

We only focus on solving the parameterised boolean equation systems, omitting the description of the other formalisms and their translations. The translation of a timed process and a property results in real-valued parameters for the equations in the parameterised boolean equation system. Such a system of equations can be reduced to a *boolean equation system* by instantiating values for the parameters; this is where real-valued parameters introduce a problem, because the uncountable infinite number of choices in universal and existential quantifiers. A boolean equation system, when it can be constructed, can finally be solved using several techniques, for example by using Gauß elimination. The solution of parameterised boolean equation systems is not defined constructively, but a given solution is shown to be correct by using a so-called *proof graph.*

A different formalism that can be used to model systems with time is the notion of a *timed automaton.* A timed automaton is essentially a finite automaton extended with multiple real-valued variables, often referred to as clocks [1]. Model checking timed automata has a similar problem with the infinite behaviour of these real-valued variables. However, for timed automata model checking questions can be solved by using a so-called *region* abstraction [2]. This abstraction relies on obtaining a finite partitioning of the real-valued domain using these regions. Another abstraction was defined for timed automata where regions are combined into zones, which results in a coarser partitioning [1]. This abstraction often works better in practice, but by providing a coarser partitioning some properties can no longer be verified.

In this thesis we present an abstraction for equation systems with real-valued parameters based on these regions as defined for timed automata. Similarly, an alternative abstraction is defined that uses zones. Both abstractions are presented as a translation of a parameterised boolean equation system with real-valued parameters to one with regions, respectively zones. For both of these translations a proof of correctness is shown that relates the solution of the result of the translation and the original parameterised boolean equation system. These abstractions only work when their real-valued parameters behave similarly to the clocks in timed automata, which require

several restrictions on the parameterised boolean equation systems.

The structure of this thesis is as follows. In Chapter 2 some preliminaries related to parameterised boolean equation systems and vector spaces are introduced. This includes the notion of a data theory to specify the sorted variables and the notion of fixed points. In Chapter 3 parameterised boolean equation systems are defined and their solution is defined by a so-called *proof graph*. In Chapter 4 timed automata are defined. In Chapter 5 the region abstraction for parameterised boolean equation systems is defined, followed by a proof of correctness. In Chapter 6 the zone abstraction are defined, again followed by a proof of correctness. In Chapter 7 an implementation in the mCRL2 language is presented that can be used to solve parameterised boolean equation systems resulting from these abstractions. In Chapter 8 several example parameterised boolean equation systems with real-valued values are abstracted, if applicable using both methods, and solved utilizing the presented implementation. Finally, our conclusion is presented in Chapter 9 and potential future work is explored in Chapter 10.

1.1 Related work

Some related work has been conducted that involved applying a similar zone abstraction to a linear process specification. The linear process specification is the result of an earlier step in the translation required to obtain the solution of a model checking question in the mCRL2 toolset. The approach taken was to translate a linear process specification with actions that refer to time and translate it to an untimed linear process specification. This way the resulting parameterised boolean equation system contains no real-valued parameters and as such can be instantiated to a boolean equation system, which can be solved whenever it is finite. The main disadvantage of this method was that it was no longer possible to verify properties with time as this information was already lost before the parameterised boolean equation system was obtained. Our method does allow us to verify properties with time, although these are still restricted in some way.

Chapter 2

Preliminaries

2.1 Lattice and fixed points

In this section the notion of fixed points, which is important for parameterised boolean equation systems, is defined, the requirements for the existence of a fixed point and the computation of a fixed point. A fixed point is defined over a so-called *lattice*, so these are introduced first.

Definition 1. A relation R on a set A is called a *partial order* if it is *reflexive*, *antisymmetric* and *transitive*. The relation R is *reflexive* if and only if for all elements $a \in A$ it holds that $(a, a) \in R$. The relation is *antisymmetric* whenever for all $a, b \in A$ it holds that if $(a, b), (b, a) \in R$ then a equals b. Finally, the relation R is *transitive* whenever for any three elements $a, b, c \in A$ it holds that if $(a, b), (b, c) \in R$ then $(a, c) \in R$.

For binary relations such as the less or equal relation between numbers, denoted by $\leq$, it is often more convenient to write it in infix notation, i.e., $d \leq e$ as opposed to $(d, e) \in \leq$.

A *lattice* is the pair of a partial order and a set, which is referred to as a partial ordered set, with one extra condition.

Definition 2 (Lattice)**.** A *lattice* $\langle A, \leq \rangle$ where A is a set and $\leq$ is a partial order on A is a lattice if and only if for all elements $a, b \in A$ the least upper bound $a \sqcup b$ and greatest lower bound $a \sqcap b$ exist. The least upper bound is the least element such that both $a \leq a \sqcup b$ and $b \leq a \sqcup b$, its dual is the greatest lower bound.

The notion of a supremum can also be extended to subsets as follows. The supremum of a subset $\bigsqcup B$, where $B \subseteq A$ is the least element $a \in A$ such that for every $b \in B$ it holds that $b \leq a$. Similarly for the infimum. When a lattice satisfies that for every subset the supremum and infimum exist it is called *complete*.

Definition 3 (Complete lattice)**.** A lattice is called *complete* if and only if the least upper bound and greatest lower bound for any subset $B \subseteq A$ exist. These bounds are denoted by $\bigsqcup B$ and $\bigsqcap B$ respectively.

For lattices over finite sets it can be shown that any lattice is also complete, but for infinite sets that is not always the case. The greater or equal to relation over the natural numbers, i.e $\langle \mathbb{N}_{\leq 100}, \geq \rangle$, is a complete lattice. The subset relation over an arbitrary set B, i.e. $\langle 2^B, \subseteq \rangle$, is another complete lattice. However, the lattice with the greater or equal to relation over all natural numbers $\langle \mathbb{N}, \leq \rangle$ is not complete as the supremum of the set $\mathbb{N}$ itself does not exist. However, it is a valid lattice as for any $a, b \in \mathbb{N}$ it holds that $a \sqcap b$ equals a if $b \leq a$ and $a \sqcap b$ equals b otherwise.

Now the notion of fixed-points can be defined. Essentially, a fixed point is a value in the domain of an (endo)-function, which is a function whose domain is equal to its co-domain, that is mapped onto itself. This can be defined as followed:

Definition 4. Let $f : A \to A$ be an (endo-)function and a a value in A then a is called a *fixed point* whenever $f(a)$ equals a.

For example, the real-valued function $f(x) = x^2$ has 1 as a fixed point, because $f(1) = 1^2 = 1$. However, a real-valued function $f(x) = x + 1$ has no fixed point, because every x is mapped to a value $x + 1$ which can never be equal to x. An interesting result, sometimes referred to as Tarski's fixed point Theorem [10], is that for *monotonic* functions over complete lattices the existence of fixed points can be guaranteed. Monotonic functions are functions over (partially) ordered sets that preserve (or reverse) the order of the elements in the set. This means that for two elements on the domain a and b; if a is greater than b then $f(a)$ is also greater than $f(b)$ if f is a monotonic function. This case is also called (monotonically) *increasing*, formally defined as:

Definition 5. Let $f : A \to A$ be a function and $\leq$ a partial order. Then f is a *increasing* function if and only if for any two elements $a, b \in A$ it holds that whenever $a \leq b$ then $f(a) \leq (b)$.

Now Tarski's fixed point theorem can be defined, which states that for increasing (or decreasing) functions over complete lattices the least and greatest fixed point exist.

Theorem 1. *Let $f : A \to A$ be an increasing function over a complete lattice $\langle A, \leq \rangle$. Then P is a non-empty set of fixed points for f and in particular P contains the least and greatest fixed points, denoted by μP and νP respectively.*

Proof. See [10]. □

This last theorem is important as it shows that the least and greatest fixed points of increasing functions exist. Finally they do not only exist, but they can be constructively defined using a (trans-) finite approximation:

Definition 6. Let $\langle A, \leq \rangle$ a complete lattice and let $f : A \to A$ be a monotonic function. Let $\bot$ the greatest lower bound of A and $\top$ the least upper bound respectively. Finally, let λ be the smallest ordinal such that the class $\{\delta \mid \delta \in \lambda\}$ has a cardinality greater than the cardinality of A. Then, the least solution to the fixed point X^λ can be computed using the following (trans-) finite approximation:

$$
\begin{aligned}
X^0 &= \bot \\
X^\delta &= f(X^{\delta-1}) && \text{For every successor ordinal } \delta \in \lambda. \\
X^\delta &= \bigsqcup_{\alpha<\delta} X^\alpha && \text{For every limit ordinal } \delta \in \lambda.
\end{aligned}
$$

For the greatest solution the initial value becomes $X^0(x) = \top$ and for every limit ordinal $\alpha \in \mu$ it is defined as $\bigsqcap_{\alpha<\lambda} X^\alpha$.

For sorts with a cardinality smaller than the cardinality of natural numbers the approximation is given by $X^\lambda = X^n$ where X^n is equal to X^{n+1} for some $n \in \mathbb{N}$. This approximation procedure for successor ordinals is also called *fixed point iteration*, due to the fact that the fixed point can be calculated by repeatedly applying the function f.

2.2 Vector spaces

The regions and zones that are introduced later on are used to partition an n-dimensional Euclidean space into a finite number of subsets. The definition that is required for this is a notion of a *convex* set. Intuitively a convex set is a set such that for every two points it is possible to define a line segment between them, such that every point on this line segment lies within the convex set. Formally it is defined as:

Definition 7. A subset of an Euclidean space $S \subseteq \mathbb{R}^n$ is *convex* if and only if for all points $x, y \in S$, $t \in \mathbb{R}$ and $0 < t < 1$ the point $(t - 1)x + ty \in S$.

2.3 Data

We assume that there is some kind of abstract data theory that describes data types and data transformations. This means that there are non-empty data sorts, typically denoted by D, E or F. There is a set $\mathcal{V}$ of sorted variables with elements $d, e, ...$ and we assume that there is some data language that is sufficiently rich to denote all relevant *data terms*. Each data sort D has an associated semantic set $\mathbb{D}$. A term t is either a variable in $\mathcal{V}$ or a function application. A function is defined by a function symbol, typically denoted by special notation `function`, and number of input sorts and a single output sort. Each function symbol has an associated semantic definition, denoted by the notation *function*. A function application is a term that consists of a function symbol and a number of terms that match its input sorts.

We use a *data environment* δ that assigns a value to the variables that can occur in a term. The application to a sorted variable d in $\mathcal{V}$ is given by $\delta(d)$, where δ is extended from a variable mapping to a mapping on terms in the standard way. The data environment can be changed by a so-called function *update*. For a given data environment δ an update is denoted by $\delta[d \leftarrow b]$ where d is a sorted variable in $\mathcal{V}$ and b is a value in its associated semantic set. The resulting data environment $\delta[d \leftarrow b]$ assigns exactly the same value for all terms that are not d, but the application of $\delta[d \leftarrow b]$ to d yields b.

Furthermore, we assume the existence of several predefined sorts. Starting with a sort Bool consisting of two distinct elements $\{true, false\}$ representing the booleans $\mathbb{B}$. For numeric values there is a sort Nat representing all natural numbers $\mathbb{N}$ and a sort Real representing the real numbers $\mathbb{R}$, which includes both the rational and the irrational domains. Furthermore we assume the existence of standard functions on these sorts, such as logical operators on booleans and arithmetic operators over numbers with their usual semantics. These operators are written in infix notation.

Remark 1. Note that for standard boolean operators, such as boolean connectives and quantification, and standard arithmetic operators, such as addition and subtraction, we use the same symbol to indicate syntactical and semantic application of these functions.

Given a sort A we also assume the existence of a complex sort List(D) that represents a list where each element in the list is an element in $\mathbb{D}$. Let $\vec{a}$ be a variable of the list sort with value $[a_0, ..., a_i]$. Let $\vec{a}_j$ for some natural number $0 \leq j < i$ denote the jth element in the list, i.e. the value a_j. Every $\vec{a}_j$ for natural numbers $0 \leq j < i$ is an element of sort A. Furthermore let $|\vec{a}|$ denote the length of the list, which in this case is equal to $i + 1$. The associated semantic set of a list sort List(D) is denoted by $\vec{\mathbb{D}}$. Semantically a finite list of length $i \in \mathbb{N}$ represents a function $f : \mathbb{N} \rightarrow \mathbb{D} \cup \bot$ such that for all natural numbers k if $0 \leq k \leq i$ it holds that $f(k) \in \mathbb{D}$ and $f(k) = \bot$ otherwise.

In the implementation chapter we see that the mCRL2 toolset provides these predefined sorts.

Chapter 3

Parameterised boolean equation systems

A *parameterised boolean equation system* can be used to encode the satisfaction relation $L \models \psi$, where L is a model specified in some formal language and ψ is a property. Each equation in such a system encodes whether a part of the property holds in a part of the model. The system of equations then encodes the complete model checking question, which gives an answer to whether the property holds in the model.

3.1 Parameterised boolean equation systems

A parameterised boolean equation is a fixed point equation that ranges over a formula [9]. Each equation has the structure $\zeta X(d : D) = \phi_X$. The left-hand side consists of a fixed point operator $\zeta \in \{\mu, \nu\}$, where μ denotes the least fixed point and ν the largest, and a (sorted) predicate variable X, taken from a sufficiently large set $\mathcal{X}$. The right-hand side of a predicate variable X is a predicate formula, denoted by ϕ_X. The domain of a predicate variable $X : D \to \text{Bool}$ is $\mathbb{D}$.

Remark 2. The symbol $=$ is used because it does not denote an equivalence relation, but rather an assignment from a predicate formula to a predicate variable.

The syntax of a predicate formula can be defined as follows.

Definition 8 (Syntax of predicate formulae)**.** The syntax of a *predicate formula* ϕ is given by the following grammar:

$$\phi \ ::= \ b \ \mid \ X(e) \ \mid \ \phi \wedge \phi \ \mid \ \phi \vee \phi \ \mid \ \forall_{d:D}.\phi \ \mid \ \exists_{d:D}.\phi$$

Here, b is a boolean term, e is a data term of sort D, X is a predicate variable and d is a data variable of sort D.

The boolean term b is used to represent any boolean expression, e.g., $m + n > 5$.

Remark 3. Negation is missing from the predicate formulae to ensure *syntactical monotonicity*. Syntactical monotonicity also ensures semantical monotonicity. This monotonicity is important to establish the existence of a fixed point solution.

A system of predicate equations can be formalized as shown below.

Definition 9 (Syntax of parameterised boolean equation systems)**.** A *parameterised boolean equation system* $\mathcal{E}$ (hereafter referred to as equation system) is defined through the grammar shown below. The symbol ε indicates the empty equation system.

$$\mathcal{E} \ ::= \ \varepsilon \ \mid \ (\mu X(d_X : D) = \phi)\mathcal{E} \ \mid \ (\nu X(d_X : D) = \phi)\mathcal{E}$$

The parameter of X is denoted by d_X. We often omit the trailing ε from example equation systems. Next, we define several properties of the predicate variables inside an equation system.

Definition 10. Let $\mathcal{E}$ be an equation system. Then predicate variables occurring at the left-hand side of each equation, these are called *bound*, are denoted by $bnd(\mathcal{E})$. Variables that are not bound are called *free*, because their value is defined outside of this equation system. The predicate variables occurring on the right-hand side of each equation are captured in $occ(\mathcal{E})$.

An equation system is *well-formed* whenever every bound predicate variable occurs at the left-hand side of (at most) one equation. We only consider well-formed equation systems in this thesis. An equation system is *closed* when there are no free variables, formally $occ(\mathcal{E}) \subseteq bnd(\mathcal{E})$.

The semantics of a predicate formula is given by an interpretation of the syntax that depends on two environments. One of these environments is called a *valuation*. Let Val denote a set of all possible functions $\eta : \mathcal{X} \to (\mathbb{D} \to \mathbb{B})$. Each function in Val denotes a valuation. This valuation assigns the result of a predicate equation, i.e. a function from data to booleans, to each predicate variable. The second environment is a data environment that assigns meanings to data terms.

The interpretation of the predicate formulae with given environments is pretty straightforward, its semantics are the same as regular boolean formulae with boolean connectives, universal and existential quantifications. The terms b and $X(e)$ both describe some boolean value. For example, the interpretation of a predicate formula $X(a) \wedge a > 5$ yields true whenever the variable a has a value in the data environment greater than 5 and $X(a)$ yields true in the valuation. However, we will see later on that these dependencies, as obtained by $X(a)$, introduce the complexity in this system of equations. Formally the interpretation can be defined as follows.

Definition 11 (Semantics of predicate formulae)**.** The interpretation of a predicate formula ϕ in the context of a data environment δ, which assigns meaning to data variables, and a valuation η, which assigns meaning to predicate variables, is defined as $[\![\phi]\!]\eta\delta$ using structural induction.

$$
\begin{array}{rl}
[\![b]\!]\eta\delta & \text{is true iff } \delta(b) \text{ is true} \\
[\![X(e)]\!]\eta\delta & \text{is true iff } \eta(X)(\delta(e)) \text{ is true} \\
[\![\phi \wedge \psi]\!]\eta\delta & \text{is true iff } [\![\phi]\!]\eta\delta \text{ and } [\![\psi]\!]\eta\delta \text{ are true} \\
[\![\phi \vee \psi]\!]\eta\delta & \text{is true iff } [\![\phi]\!]\eta\delta \text{ or } [\![\psi]\!]\eta\delta \text{ is true} \\
[\![\forall_{d:D}.\phi]\!]\eta\delta & \text{is true iff for all } e \in \mathbb{D} \text{ the interpretation } [\![\phi]\!]\eta\delta[d \leftarrow e] \text{ is true} \\
[\![\exists_{d:D}.\phi]\!]\eta\delta & \text{is true iff there is an } e \in \mathbb{D} \text{ such that } [\![\phi]\!]\eta\delta[d \leftarrow e] \text{ is true}
\end{array}
$$

The interesting part of an equation systems is its solution, which is an element of Val. The solution can be defined by an interpretation for the semantics of an equation system, which is denoted by $[\![\mathcal{E}]\!]\eta\delta$. For closed equation systems for bound predicate variables $X \in bnd(\mathcal{E})$ and values $e \in \mathbb{D}$ the solution of $[\![\mathcal{E}]\!]\eta\delta(X)(e)$ is independent of the valuation η and data environment δ used. In this case the environments can be dropped, as such the solution is given by $[\![\mathcal{E}]\!](X)(e)$. Instead of defining $[\![\mathcal{E}]\!]$, the solution can also be presented using the notion of a so-called *proof graph*, which will be introduced in the next section [7].

3.2 Proof graphs

A proof graph is a graph that contains enough information to show that a solution is correct. As mentioned earlier the solution to an equation system is given by a boolean value for each combination of a predicate variable X and a value $d \in \mathbb{D}$. A pair of a predicate variable and a value in its parameter's domain is called a *signature* and can be defined in the following way:

Definition 12 (Signatures)**.** Let $X : \mathbb{D} \to \mathbb{B}$ be a predicate variable. We say that the *signature* of X, denoted as $sig(X)$, is the product $\{X\} \times \mathbb{D}$. This concept is lifted to a set of equations $P \subseteq \mathcal{X}$ by $sig(P) = \bigcup_{X \in P} sig(X)$. For an equation system $\mathcal{E}$ it is then defined as $sig(\mathcal{E}) = sig(bnd(\mathcal{E}))$.

For these signatures we write $X(e)$ to denote a signature (X, e) to emphasize that X is an predicate variable and e is a value in the parameter of X. Consider the equation system $(\mu X(d : \text{Nat}) = d > 5)$, the tuple $X(5)$ is a possible signature of the equation X.

Every vertex in a proof graph is an element of the signature of an equation system for which the valuation yields true. Meaning that for the solution of an equation system the application of X and v yields true if and only if the signature $X(v)$ is part of its corresponding proof graph.

The edges of a proof graph are obtained from the dependencies of predicate variables on other predicate variables in the predicate formula. Every edge should satisfy the so-called *fulfilment* property. The fulfilment property states that given that these dependencies hold, which is true because these signatures are part of the proof graph, the interpretation of the predicate formula holds as well. To define this property it is necessary to be able to set values in a given environment to a specific value, this is referred to as an *update*. For any valuation η the valuations $\eta[S \mapsto \text{true}]$ and $\eta[S \mapsto \text{false}]$ are equal to η in which every signature in S has been set to true, or false respectively. This can be formally defined as:

Definition 13 (Valuation update)**.** Let η be a valuation, $S \subseteq sig(\mathcal{X})$ a set of signatures and b a boolean constant. Then a valuation update, denoted by $\eta[S \mapsto b]$, results in a valuation that is defined as:

$$(\eta[S \mapsto b])(X)(v) = \begin{cases} b & (X, v) \in S \\ \eta(X)(v) & (X, v) \notin S \end{cases} \tag{3.1}$$

Using this notation we can define the fulfilment property, as shown in equation (3.2). The *fulfilment* graph is then simply a graph where each edge satisfies this property.

Definition 14 (Fulfilment graph)**.** Given a closed equation system $\mathcal{E}$, some valuation η, some data environment δ and a graph $(V, \to)$, where $V \subseteq sig(\mathcal{E})$ and $\to \subseteq V \times V$. Then this graph is a (positive) *fulfilment graph* if and only if $\to$ satisfies that for all $X(v) \in V$:

$$[\![\phi_X]\!](\eta[sig(\mathcal{E}) \mapsto \text{false}][X(v)^\bullet \mapsto \text{true}]\delta[d_x \leftarrow v] \tag{3.2}$$

In which $X(v)^\bullet$ indicates the successor set of $X(v)$, formally $\{X'(v') \in V \mid X(v) \to X'(v')\}$.

The fulfilment graph has requirements on the vertices and edges, but nothing is stated about the infinite paths in the graph. However, the infinite paths in the graph also imply an infinite dependency on certain predicate variables. This infinite dependency relates to the definition of a largest fixed point. For equation systems this notion is more complex as the equations are ordered and this is given by the notion of a *rank*. The rank of each predicate variable can be defined as follows.

Definition 15 (Ranks)**.** A μ-block is an equation system of μ-signed equations. Likewise for a ν-block. Let $\mathcal{E} = B_1 ... B_n$ for alternating μ,ν-blocks $B_1, ..., B_n$ denote an equation system. The *rank* of a bound predicate variable $X \in bnd(\mathcal{E})$, denoted as $rank_{\mathcal{E}}(X)$ is equal to the number of alternating μ-blocks and ν-blocks, formally defined as:

$$\text{for all } (\zeta X(d : D) = \phi) \in B_i = \begin{cases} i & \text{if } B_1 \text{ is a } \mu\text{-block} \\ i - 1 & \text{otherwise} \end{cases} \tag{3.3}$$

For predicate variables that occur in $\mathcal{E}$ but are not bound $X \in occ(sig) \backslash bnd(\mathcal{E})$ the rank $rank_{\mathcal{E}}(X)$ equals minus one.

Note that predicate variables bound by a largest fixed point operator have an even rank and the variables bound by a smallest fixed point operator an odd rank. Now the proof graph property can be introduced. A proof graph then becomes a fulfilment graph that satisfies this additional property, as stated in the following definition.

Definition 16. A *proof graph* is a positive fulfilment graph $(V, \rightarrow)$ for an equation system $\mathcal{E}$ and valuation η where for all infinite paths $Z_0(v_0) \rightarrow Z_1(v_1) \rightarrow Z_2(v_2) \rightarrow \cdots$ the minimal rank of all predicate variables that occur infinitely often is even.

A proof graph might contain redundant vertices and edges. To prevent these the notion of a *minimal* proof graph can be introduced.

Definition 17. A proof graph is minimal w.r.t. a proof graph G and a node v in G if and only if it is a sub-graph of G, includes v and there is no sub-graph of fewer nodes or edges that is a proof graph and includes v.

Finally, the main reason for introducing proof graphs is captured in the following theorem. It captures the relation between the solution of an equation system and the corresponding proof graph.

Definition 18. Given a closed equation system $\mathcal{E}$ then for every bound predicate variable $X \in bnd(\mathcal{E})$ and all $v \in \mathbb{D}$ the interpretation $[\![\mathcal{E}]\!](X)(v)$ is true if and only if a proof graph $G = (V, \rightarrow)$ for $\mathcal{E}$ exists for which $X(v) \in V$.

In the next section we will introduce an example equation system and show its corresponding proof graph.

3.3 Example

Consider the following example equation system.

Example 1. Consider the following equation system $\mathcal{E}$:

$$\begin{aligned}\mu X(d : \text{Nat}, x : \text{Nat}, y : \text{Nat}) = d \approx 1 \wedge \\ (\exists_{t:\text{Nat}}.(2 < x + t \leq 5 \wedge X(1, 0, y + t)) \\ \vee \exists_{t:\text{Nat}}.(2 < x + t \leq 5 \wedge y + t \geq 10))\end{aligned}$$

We are interested in the solution for $X(1, 0, 0)$. Note that the symbol $\approx$ is used to denote that sorted variable d has a value assigned by the data environment that is equal to one as opposed to variable d being equal to one which is not true.

The signatures of this equation system are $X(e, x', y')$ where e, x', y' are natural numbers. The following proof graph encodes the solution for this equation system, proving that the solution of $X(1, 0, 0)$ yields true.

Example 2. A minimal proof graph for $X(1, 0, 0)$ is:

$$X(1, 0, 0) \longrightarrow X(1, 0, 3) \longrightarrow X(1, 0, 8)$$

The fulfilment property holds for all transitions and there are no infinite paths.

The tools of the mCRL2 toolset solve these equation systems by instantiating them to a boolean equation system. A boolean equation system is a special parameterised boolean equation system where every domain of a predicate variable is empty. This resulting boolean equation system can be solved using so-called *Gauß* elimination. The instantiation to a boolean equation system is obtained by recursively defining all occurring predicate variables starting from the initial equation in such a way that the boolean equation X_e gives the solution for $X(e)$ in the parameterised boolean equation.

Example 3. Consider the same equation system as in Example 1. Start by instantiating the initial variable $X(1,0,0)$ and defining the right-hand side. The existential quantification over t must be enumerated over all possible values of t, whenever t satisfies $2 < t \leq 5$, resulting in:

$$\mu X_{1,0,0} = X_{1,0,3} \vee X_{1,0,4} \vee X_{1,0,5}$$

In theory it should consider all values of t, but in practice the tool-set is capable of proving that $t > 5$ can never satisfy $t \leq 5$ using induction. Now the occurring variables $X_{1,0,3}$, $X_{1,0,4}$ and $X_{1,0,5}$ must be defined. This exploration is repeated until all equations that occur have an equation that defines their right-hand side. This means that for each $X_{d,x,y}$ there is an equation that defines its right-hand side, resulting in:

$$\begin{aligned}
\mu X_{1,0,0} &= X_{1,0,3} \vee X_{1,0,4} \vee X_{1,0,5}\\
\mu X_{1,0,3} &= X_{1,0,6} \vee X_{1,0,7} \vee X_{1,0,8}\\
\mu X_{1,0,4} &= X_{1,0,7} \vee X_{1,0,8} \vee X_{1,0,9}\\
\mu X_{1,0,5} &= \text{true}\\
\mu X_{1,0,6} &= \text{true}\\
\mu X_{1,0,7} &= \text{true}\\
\mu X_{1,0,8} &= \text{true}\\
\mu X_{1,0,9} &= \text{true}
\end{aligned}$$

Note that the resulting boolean equation system at least contains the dependencies from the proof graph, but in this case the enumeration results in more equations. Also note that equations with a right-hand side equal to true occur in the proof graph without any dependencies, as shown by $X(1,0,8)$.

Solving equation systems that can be instantiated to a finite boolean equation system is always possible. However, we motivate the need for an abstraction using a real-valued variant of the equation system presented in Example 1.

Example 4. Consider the equation system from Example 1 again, but with the parameters x, y and bound variable t having the Real sort. This change results in the equation system presented below.

$$\begin{aligned}
\mu X(d:\text{Nat}, x:\text{Real}, y:\text{Real}) = & d \approx 1 \wedge\\
& (\exists_{t:\text{Real}}.(2 < x + t \leq 5 \implies X(1,0,y+t))\\
& \vee \exists_{t:\text{Real}}.(2 < x + t \leq 5 \wedge y + t \geq 10))
\end{aligned}$$

In this case, because the $\mathbb{R}$ domain is uncountable infinite, it is not possible to enumerate over all possible values of variable t and instantiating the boolean equations $X_{1,0,y+t}$ whenever t satisfies $2 < t \leq 5$, because there are infinitely many such t.

This example shows that we require some finite abstraction of the real-valued domain such that it is possible to instantiate a boolean equation system. In the next chapter we will define timed automata, because a similar problem exists with their real-time behaviour. For timed automata the region and zone abstractions can already be applied in practice and as such they offer leverage to the abstraction that we will present later on.

Chapter 4

Timed automata

In this chapter we introduce timed automata as defined in [1]. Timed automata are more restricted in their behaviour than the timed process algebra of the mCRL2 toolset. However, model checking questions can already be solved in practice for timed automata using region and zone abstractions. As such, their restrictions offer insight in the type of restrictions required to apply similar region and zone abstractions on equation systems.

4.1 Timed automata

A timed automaton has a finite set of *locations*, with one location marked as initial. It also has a set of real-valued variables, often referred to as clocks. Each location can have a so called *location invariant* that should not be violated. This invariant constrains the values that each clock might have in that location. The timed automaton also has a set of *input symbols*, this is the alphabet. A *switch* represents a transition between two locations. Each switch has an input symbol, an *enabling condition* and a subset of clocks that are reset.

The input symbol relates to the language that this timed automaton can accept. A transition can only be taken when the enabling condition is fulfilled. Each clock in the subset of clocks that is reset is set to zero upon taking the switch. Switches occur instantaneously, however time can progress inside states and this is referred to as *idling*. The value of each clock equals the time elapsed since the last reset, which implies that every clock progresses with the same speed.

Example 5. In Figure 4.1 an example automaton is shown with locations $\{l_0, l_1\}$, where l_0 is the initial location. It models a system where input b always occurs between one to two time units after input a occurred. There is one clock x, initially set to zero. Switch $l_0 \rightarrow l_1$ sets the clock x to zero when an a occurs. In location l_1 the invariant $x \leq 2$ requires that clock x never exceeds two. On the other hand the enabling condition of switch $l_1 \rightarrow l_0$ requires that this switch can be taken whenever x is greater than one. So combined in location l_1 the time passes at least 1 second and at most 2 seconds before a b must occur.

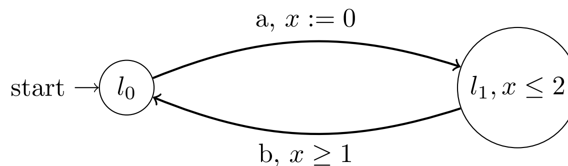

Figure 4.1: An example timed automaton

The formal definition for a timed automaton is shown in Definition 21. It requires the definition of a *clock constraint* $\Phi(C)$, where C is a set of clocks.

Definition 19 (Syntax of clock constraints)**.** For a set of clocks C, the set of clock constraints $\Phi(C)$ is defined by the following grammar:

$$b_c \ ::= \ x \geq c \ \mid \ c \geq x \ \mid \ x < c \ \mid \ c < x \ \mid \ b_c \wedge b_c$$

Where x is a clock in C and c is a constant in $\mathbb{N}$, the set of natural numbers.

Remark 4. Having natural constants instead of rational constants is not a restriction, because every rational constant in a clock constraint can be converted to an natural constant by multiplying with the lowest common divisor of all of constants occurring in all clock constraints.

This grammar for clock constraints is called *diagonal free* as opposed to an alternative grammar for clock constraints that additionally allows constraints of the structure $x - y < c$ and $x - y \leq c$, where x and y are clocks from C and c is a natural constant. Another important definition is the *clock interpretation*, which is an environment v where each clock in C is given some value in the positive real-valued domain, denoted as $\mathbb{R}$.

Definition 20 (Clock interpretation)**.** Given a set of clocks C, a clock interpretation $v : C \to \mathbb{R}$ assigns a value to each clock.

A clock interpretation v over a set of clocks C satisfies a clock constraint $\phi \in \Phi(C)$ if and only if ϕ evaluates to true according to the values that were assigned by v, this is denoted as $v \models \phi$. For a set of clocks $X \subseteq C$ we write $v[X \leftarrow 0]$ for the clock interpretation, where for each clock $x \in X$ its value is set to zero and all other clocks are given by v, i.e. $(v[X \leftarrow 0])(x)$ is $v(x)$ if $x \notin X$ and zero otherwise. The notation $v + c$, for some $c \in \mathbb{R}$, is used to denote a new interpretation where every clock value $v(x)$ is increased by c, defined as $\forall_{x \in C}(v + c)(x) = v(x) + c$.

Definition 21 (Timed automaton)**.** A timed automaton A is a tuple $\langle L, l_0, C, \Sigma, \to, I \rangle$

- The finite set of locations L.
- One initial location $l_0 \in L$.
- The set of real-valued variables C, also referred to as clocks.
- The alphabet of input symbols Σ.
- The set of switches $\to \subseteq L \times \Phi(C) \times \Sigma \times 2^C \times L$. A switch $(l, \phi, \sigma, \lambda, l') \in \to$ can be written as $l \xrightarrow{\phi,\sigma,\lambda} l'$. A switch $l \xrightarrow{\phi,\sigma,\lambda} l'$ indicates that from location l a switch can be made to l' emitting the input symbol σ and resetting the clocks in λ, whenever ϕ is satisfied by the current clock interpretation and the invariants of l and l' are not invalidated by the progression of time.
- The function $I : L \to \Phi(C)$ assigns invariants to locations.

Verification conceptually operates on the semantics of a timed automaton, which is given in terms of a *timed transition system.* Therefore, these will be introduced in the next section.

4.2 Timed transition systems

A timed transition system is a labelled transition system extended with the notion of a global time [8]. A labelled transition system consists of a (possibly infinite) set of states and a transition relation. There is one initial state and each transition is labelled with an action.

In a timed transition system each transition contains the global time at which it occurred. This replaces the notion of multiple clocks. However, relative time can still be obtained by using real-valued variables inside states. Time can also progress inside a state without performing an action. This is called *idling* and is denoted by a special arrow $\rightsquigarrow$. The formal definition of a timed transition system is given below.

Definition 22 (Timed transition system)**.** A timed transition system T is a tuple $\langle S, s_0, Act, \rightarrow, \rightsquigarrow \rangle$.

- The (possibly infinite) set of states S.
- One state $s_0 \in S$ is the initial state.
- The set of actions Act.
- The transition relation $\rightarrow \subseteq S \times Act \times \mathbb{R}_{>0} \times S$. A transition $(s, a, t, s') \in \rightarrow$ can be written as $s \xrightarrow{a}_t s'$. The expression $s \xrightarrow{a}_t s'$ states that a transition is made from s to s' by executing action a at time t.
- The idle relation $\rightsquigarrow \subseteq S \times \mathbb{R}_{>0}$. The predicate $s \rightsquigarrow_t$ expresses that in state s it is possible to idle up to and including time t.

The set of positive real numbers is denoted by $\mathbb{R}_{>0}$. For valid timed transition systems two additional properties must be satisfied. The first property states that time must always progress in consecutive transitions.

Definition 23. Let T be a timed transition system. Then the *progress* property states that for all transitions $s \xrightarrow{a}_t s' \xrightarrow{a'}_{t'} s''$ it must hold that $t' > t$.

The second property states that it is always possible to idle to a time earlier than the next action or idle transition.

Definition 24. Let T be a timed transition system. Then the *density* property states that if there is a transition $s \xrightarrow{a}_t s'$ or if it possible to idle $s \rightsquigarrow_t$ then it is possible to idle $s \rightsquigarrow_{t'}$ with $0 < t' < t$.

Timed transition systems are a very operational way of describing the behaviour of a system. With data types the transition system can already be infinite in size, but at least they can be enumerated in some way. By introducing real-valued variables, which is exactly what time is, there can be infinitely many values between any two real values t_1 and t_2.

4.3 Semantics of timed automata

The semantics of a timed automaton can be defined by a timed transition system that defines when the timed automaton can do a transition, which emits an input symbol when this transition is taken.

Definition 25 (Semantics of timed automata)**.** Let $A = \langle L, l_0, C, \Sigma, \rightarrow_A, I \rangle$ be a timed automaton. Then $T_A = \langle S, s_0, Act, \rightarrow_T, \rightsquigarrow \rangle$ is a timed transition system associated to A. The timed transition system T_A is defined as follows.

- The set of states S is defined as $\{(l, \upsilon, T) \mid l \in L, \upsilon \in \mathbb{R}^C, T \in \mathbb{R}\}$, where υ is the interpretation of clocks and T is a new variable referred to as the last action time.
- The initial state s_0 is given by the state $(l_0, 0^C, 0)$, where 0^C is the valuation in which every clock is zero and the last action time is also zero.
- The action set Act is equal to the set of input symbols Σ.
- For every switch $l \xrightarrow{\phi, \sigma, \lambda} l' \in \rightarrow_A$ there are transitions $(l, \upsilon, t) \xrightarrow{\sigma}_{t'} (l', \upsilon', t') \in \rightarrow_T$ such that the following conditions hold:
  - The progress condition is satisfied: $t' > t$.
  - The enabling condition is satisfied: $\upsilon + (t' - t) \models \phi$.

  - The location invariant of l is never invalidated, i.e. $\forall_{u \in \mathbb{R}}(u \leq t' - t) \implies (v + u) \models I(l)$. For linear constraints this is the same as $v \models I(l)$ and $v + (t' - t) \models I(l)$.
  - Each clock in λ has been reset and all other clocks have been increased by $t' - t$ in the successor state $v' = (v + (t' - t))[\lambda \leftarrow 0]$.
  - The location invariant $I(l')$ is satisfied by v', thus $v' \models I(l')$.

- It is possible to idle in $(l, v, t) \in S$ as long as the location invariant $I(s)$ is satisfied. This means that $\forall_{u \in \mathbb{R}_{>0}} \forall_{t \in \mathbb{R}}(v + (t' - t)) \models I(l) \wedge u < t \implies (l, v, t) \rightsquigarrow_u$.

Next, we will show a part of the timed transition system for the timed automaton presented in Figure 4.1.

Example 6. Consider the timed automaton presented in Figure 4.1. The following figure shows a part of the transition system that encodes its semantics. Note that this is only a part of the transition system, because the actual transition system has an infinite number of states and transitions.

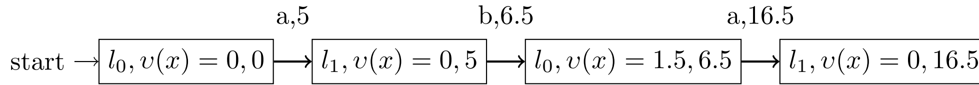

Figure 4.2: Example transitions of the timed automaton in figure 4.1

The initial state is $(l_0, 0, 0)$. Here, the time can idle to infinity, as there are no invariant defined for l_0. Assume it only idles to time five and then the a transition is taken. Now x has been reset and the last action time becomes five. Now it is possible to idle to two, but it only idles for 1.5 and then it takes transition b. Now after some time the a transition is taken again and the clock x is reset to zero and the last action time is updated. This illustrates that due to progress and global time the transition system is an infinite chain of increasing time.

The example shows that the timed transition system can have an infinite uncountable number of extended states. This also means that it is not possible to do explicit model checking on such a timed transition system directly. For timed automata a so-called *region*-equivalence has been defined where extended states are region-equivalent when their states induce equivalent runs [2]. A run is a sequence of actions in the transition system. The resulting region automaton is finite and can be used to answer model checking questions over the original automaton. In the next chapter we will show how a region abstraction can be defined for equation systems which have similar restrictions on their time behaviour as timed automata.

Chapter 5

Region equivalence for equation systems

The real-valued variables in equation systems are used to encode timed behaviour in the same way as clocks are used in timed automata. Recall that clock constraints had to be linear conditions, i.e. conditions where a clock's interpretation is compared with a natural constant. For the equation systems some constraints are defined such that conditions over real-valued variables are also linear. Furthermore the notion that every clock progresses with the same speed and the notion of clock resets must be incorporated for the behaviour of real-valued variables in equation systems as well. Now we define a region-equivalence similar to the one defined for clock interpretations in [2], followed by a detailed description of the restrictions required for equation systems with real-valued parameters and the translation itself.

For the region-equivalence, the integral part of each real-valued variable is required to determine whether or not a given timing constraint is met. The real-valued variable's fractional parts, and in particular their ordering, are required to decide which clock will change its integral part first. The latter is important for the ordering of allowed transitions, consider the following example:

Example 7. Given two real-valued variables x, y with values between zero and one in some equation. Then the timing constraint $(x = 1)$ can be met followed by a timing constraint $(y = 1)$ depending on whether the current data interpretation satisfies $x < y$. If $x < y$ holds then the timing constraint $(y = 1)$ is met first, because each real-valued variable increases its value with the same speed.

The integral parts of each clock can grow infinitely large by letting time progress. However, note that when a clock is never compared with a constant greater than c, then letting time progress above this constant does not influence the decision of any timing constraint. In Example 7 when there are no other constraints and x exceeds one then letting time progress will never satisfy $(x = 1)$ again. Similarly when y exceeds one or when both exceed one.

For timed automata the region-equivalence is first defined over clock interpretations and then extended to extended states. The notion of a clock interpretation is not present for timed equation systems. However, by changing the parameters in these timed equation systems and splitting the data environment the connection to clock interpretations can be made. Note that this means that we do not consider arbitrary real-valued parameters, but only real-valued parameters with semantics that are very similar to clocks in timed automata.

For convenience, all real-valued parameters are grouped into a single parameter list $\vec{u}$, which can be represented by a sort List(Real). Additional requirement are specified such that the same element in $\vec{u}$ refers to the same real-valued parameter in every equation, as such each predicate variable has the same $\vec{u}$ parameter. This requirement makes the variable $\vec{u}$ similar to the set of clocks for a timed automaton. For example, consider the predicate equation $X(d : \text{Real}, e : D, f : \text{Real})$. To fit the restrictions it can be changed into $X(e : D, \vec{u} : \text{List(Real)})$, where $\vec{u} = [d, f]$.

Interpreting a predicate equation is done in the context of a data environment δ. However, it is possible to consider this data environment as the combination of a timed data environment δ_t, that assigns a value to the $\vec{u}$ parameter, and another data environment δ_d, that assigns values to the other terms, such that δ equals their *disjunct union* $\delta_t \uplus \delta_d$.

Definition 26. Given two functions $f : A \to C$ and $g : B \to C$ for disjoint sets A and B, then let $f \uplus g$ denote the *disjunct union* of f and g. Formally defined as, for all $a \in A \cup B$:

$$(f \uplus g)(a) = \begin{cases} f(a) & a \in A \\ g(a) & a \in B \end{cases} \tag{5.1}$$

For every $i \in \mathbb{N}$ and $0 \leq i < k$ a constant $c_{\vec{u}_i}$ is defined to be equal to the highest constant that the variable $\vec{u}_i$ is compared to in a given equation system. It is not sufficient to have the highest constant $c_{\vec{u}_i}$ per equation in the equation system because the exact value for variables above the constant is unknown and it is unclear what value it should have when this variable is passed as an argument to a different predicate variable that compares it with a higher constant.

We assume that every element in $\vec{u}$ is used in some constraint. Considering the real-valued parameters as a list also means that the timed data environment δ_t assigning values for the variables in $\vec{u}$ can be viewed similar to a clock interpretation over a set of clocks C. Now, let us formalize the notion of a region equivalence between timed data interpretations.

Definition 27. For any value $x \in \mathbb{R}$. Let $fr(x)$ denote its fractional part and let $\lfloor x \rfloor$ denote its integral part, such that $x = fr(x) + \lfloor x \rfloor$.

Next, we can define when two real data environments δ_t and δ'_t are region-equivalent.

Definition 28 (Region equivalence)**.** Given a vector $\vec{u}$ and for every $i \in \mathbb{N}$ such that $0 \leq i < |\vec{u}|$ the value of $c_{\vec{u}_i}$. Then, the *region-equivalence* $\sim$ is defined over the set of all real data environments; $\delta_t \sim \delta'_t$ if and only if all of the following conditions holds:

- For all $0 \leq i < |\vec{u}|$, either $\lfloor \delta_t(\vec{u}_i) \rfloor$ and $\lfloor \delta'_t(\vec{u}_i) \rfloor$ are the same, or both $\lfloor \delta_t(\vec{u}_i) \rfloor$ and $\lfloor \delta'_t(\vec{u}_i) \rfloor$ exceed $c_{\vec{u}_i}$.
- For all $i, j \in \mathbb{N}$ such that $0 \leq i, j < |\vec{u}|$, with $\delta_t(\vec{u}_i) \leq c_{\vec{u}_i}$ and $\delta_t(\vec{u}_j) \leq c_{\vec{u}_j}$, $fr(\delta_t(\vec{u}_i)) \leq fr(\delta_t(\vec{u}_j))$ if and only if $fr(\delta'_t(\vec{u}_i)) \leq fr(\delta'_t(\vec{u}_i))$.
- For all $0 \leq i < |\vec{u}|$ with $\delta_t(\vec{u}_i) \leq c_{\vec{u}_i}$, $fr(\delta_t(\vec{u}_i))$ equals zero if and only if $fr(\delta'_t(\vec{u}_i))$ equals zero.

As the relation $\sim$ is an equivalence relation on timed data interpretations then it also possible to split the set of timed data environment into *equivalence classes* [1]. This notion is defined as follows.

Definition 29 (Region)**.** Let δ_t be a timed data environment. Then $[\delta_t]$ denotes the equivalent class of δ_t, this results in the set of timed data environments that are equivalent, formally $\{\delta'_t \mid \delta'_t \sim \delta_t\}$. This equivalence class, or set of timed data environments, is called a *region*.

However, it sometimes is more convenient to view the region equivalence as a partitioning on the n-dimensional space of real values, where n is equal to $|\vec{u}|$. This view is used when it is clear that a specific value is taken from a region, such as a list $\vec{v} \in R$, where R is a region. This view can be achieved by interpreting a region R as a set of values that are the result of applying a timed data environment $\delta_t(\vec{u})$ for all timed data environment $\delta_t \in R$, formally $\{\delta_t(\vec{u}) \mid \delta_t \in R\}$. In this view the distinction of a timed data environment and a data environment for other parameters is no longer required. The following example illustrates the notion of equivalence classes.

Example 8. Consider an example of a variable $[x, y]$, such that c_x equals 2 and c_y equals 1. Each area represents a set of two dimensional vectors that belong to the same region.

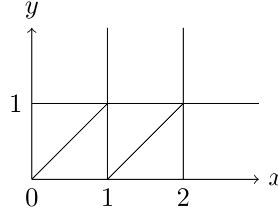

Figure 5.1: The regions for two clocks $[x, y]$, with $c_x = 2$ and $c_y = 1$.

There are six point constraints, two diagonal constraints and twelve line segments. Each region either defines the area in a right-angled triangle, a line segment, a point or the area above c_x and c_y. Note that for two region-equivalent timed data environments $\delta_t \sim \delta'_t$ it holds that for every linear constraint over the variables in $\vec{u}$ then δ_t satisfies it if and only if δ'_t does [2]. The representation of a region is generalized into n-dimensions by the following definition.

Definition 30 (Representation of regions)**.** The sort Region over a real-valued list variable $\vec{u}$ is used to represent regions by specifying:

1. For every $0 \leq i < |\vec{u}|$, one constraint from the following set
   - The variable $\vec{u}_i$ is equal to some natural constant c, formally $\{\vec{u}_i = c \mid c = 0, 1, ..., c_{\vec{u}_i}\}$.
   - The variable $\vec{u}_i$ has a value between $c - 1$ and c for some natural constant c, formally $\{c - 1 < \vec{u}_i < c \mid c = 1, ..., c_{\vec{u}_i}\}$.
   - The clock exceeds $c_{\vec{u}_i}$, formally $\{\vec{u}_i > c_{\vec{u}_i}\}$.
2. For every pair of real-valued variables $\vec{u}_i$ and $\vec{u}_j$ that both have constraints $c - 1 < \vec{u}_i < c$ and $d - 1 < \vec{u}_j < d$ in (1) for some natural constants c and d, whether $fr(x)$ is less than, equal to, or greater than $fr(y)$.

Note that there can be only a finite number of regions, bounded by the number of constraints and fractional orderings possible. This region representation is presented by the short-hand notation $[0 < x < y < 1]$, that contains both the integral constraints and the fractional ordering of $fr(x)$ and $fr(y)$.

When time progresses inside the system the value of each real-valued variable $\delta_t(\vec{u}_i)$ increases, ignoring clock resets for now, and eventually they might belong to a different region. For example a region over two clocks defined by $[x = 0; 0 < y < 1]$ will instantaneously change into a region $[0 < x < y < 1]$ where x lies between zero and one and the fractions are ordered by $fr(x) < fr(y)$. This time progress induces a *time-successor* relation between regions that can be reached by letting time progress in a clock interpretation belonging to some region.

For a list value $\vec{v}$ we introduce the following semantic operation $\vec{v} + t$ to indicate a list where every value in $\vec{v}$ is increased by t. Formally, given a list $\vec{v}$ then in $\vec{v}+t$ for every $i \in \mathbb{N}$ the resulting vector has values $(\vec{v} + t)_i = \vec{v}_i + t$. This notation is then also introduced as syntax in a predicate formula for a list variable $\vec{u}$ such that $\delta_t(\vec{u} + t)$ yields $\delta_t(\vec{u}) + t$.

Definition 31 (Time-successor)**.** A region R' over a real-sorted list variable $\vec{u}$ is a *time successor* of a region R if and only for all values $\vec{v} \in R$ there exists a $t \in \mathbb{R}$ such that $\vec{v} + t \in R'$.

Note that this defines a transitive relation, and it induces a *set of regions* reachable by this relation. The set of regions can also be determined by taking a specific value $\vec{v} \in R$ and letting time progress for some variable $t \in \mathbb{R}$, i.e. $\vec{v} + t$, and marking every region such that $\vec{v} + t$ lies within that region [2]. Furthermore it must be possible to decide whether a region satisfies a constraint similar to a clock constraint for timed automata.

For a real-valued vector we define the following timed constraints.

Definition 32 (Syntax of timed constraints)**.** Let $\vec{u}$ be a real-valued list variable. Then a timed constraint over variable $\vec{u}$ is defined by the following grammar:

$$b_u \ ::= \ \vec{u}_i \geq c \ \mid \ c \geq \vec{u}_i \ \mid \ \vec{u}_i < c \ \mid \ c < \vec{u}_i$$

Where c is natural constant.

Note that b_u defines a linear constraint and as such the observation that two region equivalent timed data interpretations both satisfy this constraint or neither do still holds. Similar to clocks in timed automata it should also be possible to reset real-valued variables a number of variables which corresponds to a resetting a number of variables in a region. Given a region R over a list variable $\vec{u}$ the following functions are defined:

- The time-successors function, denoted by $\Uparrow R$, results in the set of regions for R, i.e the regions that can be reached from every $\vec{v} \in R$ such that there exists a $t \in \mathbb{R}$ for which $\vec{v} + t \in \Uparrow R$.
- The reset function, denoted by $R[\lambda \leftarrow 0]$ where $\lambda \subseteq 2^{\mathbb{N}}$ is a set of indices is exactly the same as the reset operation defined for timed automata. Formally it results in a region such that for every variable u_i for $i \in \lambda$ the resulting region satisfies $(\delta_t(\vec{u}_i) = 0)$ and the ordering of $fr(\vec{u}_i)$ is undefined.
- The satisfies function, denoted by $R \models f$ where f is an expression of grammar b_u, yields true if every data environment in the region satisfies the given timing constraint and yields false otherwise. Formally defined as $\forall_{\vec{v} \in R}.\delta[\vec{u} \leftarrow \vec{v}](f)$. Which by the observation of linear constraints for region equivalent timed data environments is the same as $\exists_{\vec{v} \in R}.\delta[\vec{u} \leftarrow \vec{v}](f)$.

In the next section a translation is defined from an equation system with real-valued parameters into an equation system where the real parameters are represented by regions. As the number of regions is finite, this resulting equation system can be solved using an instantiation to a boolean equation system given that the number of values instantiated for variables in δ_d are finite as well.

5.1 Translation

For convenience we only consider equation systems in *standard recursive form*. Any equation system can be written as an equation system in standard recursive form by applying the Tseitin transformation [11]. For equation systems the ranks and fixed point operators of the introduced variables must be same as where the predicate sub-formula originated. This ensures that the solution for the predicate variables that occur in derived and original equation systems stay the same. The standard recursive form is defined as follows:

Definition 33 (Standard recursive form)**.** The right-hand side of a predicate equation in standard recursive form is given by:

$$\begin{aligned} \phi_s \ ::= \ & \bigvee_{i \in I} \exists_{e_i : E_i}.(f_i(e_i, d) \wedge X_i(g_i(e_i, d)))) \\ \mid \ & \bigwedge_{i \in I} \forall_{e_i : E_i}.(f_i(e_i, d) \implies X_i(g_i(e_i, d))) \end{aligned}$$

The conjunction, or disjunction, over the set I is used to specify a conjunction, or disjunction, over multiple predicate formulae. The function $f_i(e_i, d)$ is a boolean function and $g_i(e_i, d)$ is a function that maps to the sort of the predicate variable's X_i parameter. An equation system is in standard recursive form when every equation has a right-hand side of grammar ϕ_s.

We do not consider every equation system in standard recursive form for our translation, but require additional restrictions. For example, timing constraints on real-valued variables are only allowed to be linear, similar to the type of clock constraints allowed for diagonal free timed automata. To explicitly define the elapsing of time of each real-valued variable they must always occur in the context of a bound real-valued variable t. This variable is used to encode time-elapse similar to time-elapse for clock interpretations.

The set of formulae allowed over real-valued variables is defined as shown below.

Definition 34 (Syntax of time-elapse constraints)**.** Let $\vec{u}$ be a real-valued list variable. Then linear time-elapse constraints over $\vec{u}$, denoted by b_t, are defined by the following grammar:

$$b_t \ ::= \ \vec{u}_i + t < c \ \mid \ \vec{u}_i + t \leq c \ \mid \ \vec{u}_i + t > c \ \mid \vec{u}_i + t \geq c$$

Where t a free real-valued variable and c is a natural constant.

Definition 34 defines the constraints present in the original equation systems and the timing constraints defined in Definition 32 are the type of constraints present over regions. Now we can define the grammar for equation systems that can be abstracted using our region translation.

Definition 35 (Restrictions for region translation input equation systems)**.** Let $\vec{u}$ be a real-valued list variable. Then ϕ_t denotes the grammar for conjunctive and disjunctive combinations of timing constraints, defined as:

$$\phi_t ::= b_t \ \mid \ \phi_t \wedge \phi_t \ \mid \ \phi_t \vee \phi_t$$

Let $\phi_R^{\vee}$ and $\phi_R^{\wedge}$ be the grammar of existential and universal quantifications, defined as:

$$\begin{aligned}
\phi_R^{\vee} \ &::= \ \exists_{e_i:E_i}.\exists_{t:\text{Real}}.(b \wedge \phi_t(\vec{u}, t) \wedge X_i(g_i(e_i, d), h_i(t, \vec{u}, \lambda))) \\
\phi_R^{\wedge} \ &::= \ \forall_{e_i:E_i}.\forall_{t:\text{Real}}.((b \wedge \phi_t(\vec{u}, t)) \implies X_i(g_i(e_i, d), h_i(t, \vec{u}, \lambda)))
\end{aligned}$$

Here, b is a boolean term over variables e_i and d. The timing constraint ϕ_t is a condition over parameter $\vec{u}$ and the previously free variable t is now bound by a quantifier. The function $h_i(t, \vec{u}, \lambda)$ is a function that updates each value in $h_i(t, \vec{u}, \lambda)$ independently. The application of $h_i(q, \vec{v}, \alpha)$, for concrete values $\vec{v} \in \vec{\mathbb{R}}$, $q \in \mathbb{R}$ and $\alpha \subseteq 2^{\mathbb{N}}$, yields a value $\vec{w}$ such that for every $0 \leq i < |\vec{u}|$ the value $\vec{w}_i = \vec{v}_i + t$ if $i \in \alpha$ and $\vec{w}_i = 0$ otherwise.

The grammar of each right-hand side of a predicate formula, denoted by ϕ_R, is defined as:

$$\phi_R \ ::= \ \bigvee_{i \in I} \phi_R^{\vee}(i) \ \mid \ \bigwedge_{i \in I} \phi_R^{\wedge}(i)$$

Finally, the input equation system $\mathcal{E}$ is a closed equation system where every predicate equation structure $X(d : D, \vec{u} : \text{List(Real)}) = \phi_R$. Note that each $\vec{u}$ requires a list of the same length and each element in $\vec{u}$ represents the same real-valued variable.

Note that an implication is not part of the grammar defined for predicate equations. However, for predicates formulas P and Q an implication $P \implies Q$ can be rewritten to $\neg P \vee Q$. If the premise P does not contain predicate variable it it possible to define a predicate formula P' that always yields the negation of P without introducing negations such that the result $P' \vee Q$ preserves monotonicity.

Note that an equation system satisfying these restrictions can be interpreted by a data environment $\delta = \delta_t \uplus \delta_d$ such that δ_t assigns values for its parameter $\vec{u}$ and δ_d assigns values for other data variables. With all these constraints the real-valued variables in the equation systems model timing behaviour similar to the time behaviour in timed automata.

Finally, note that for arbitrary equation systems it is not obvious what restrictions it should satisfy to be able to apply a Tseitin transformation to obtain an equation system that satisfies the restrictions as defined in Definition 35.

Remark 5. Although an equation systems with real-valued parameters can be restricted in such a way that these parameters are used to model time similar to clocks in timed automata, it is not exactly the same. There is some similarity between the relation of a timed automaton and its timed transition system and between a parameterised boolean equation system and its instantiated boolean equation system in the infinite "behaviour". However, in an equation systems the "time" semantics are encoded in each predicate equation itself and there are no different interpretation for time and "other" behaviour.

Now we can define a translation to a *region equation system* that models the same timing behaviour using regions.

Definition 36 (Region equation system)**.** Let $\mathcal{E}$ be an equation system that satisfies the restrictions as defined in Definition 35. For the functions on regions we define the following syntax:

- The function `time_successors` : Region $\rightarrow$ List(Region) takes a region parameter r and returns a list of regions that are the time-successor of r. Formally the interpretation $[\![\texttt{time_successors}(r')]\!]\eta\delta$ is equal to *time_successors*$(\delta(r))$, which is defined as $\Uparrow(\delta(r))$.
- The function `is_successor` : Region $\times$ Region $\rightarrow$ Bool takes two region parameters r' and r as input and returns whether $r' \in \Uparrow r$. Formally the interpretation $[\![\texttt{is_successor}(r', r)]\!]\eta\delta$ equals *is_successor*$(\delta(r'), \delta(r))$, which is defined as $\delta(r') \in$ *time_successors*$(\delta(r))$.
- The function `satisfies` : Region $\times$ Expression $\rightarrow$ Bool takes a region parameter r and an expression parameter e of grammar b_u. This operation returns true if and only if for all $\delta_t \in r$ the condition of e is satisfied. Formally, *satisfies*$(\delta(r), \delta(e))$ is defined as $\delta(r) \models \delta(e)$.
- The function `reset` : Region $\times$ Bool^{Nat} $\rightarrow$ Region takes a region parameter r and parameter λ for the set of indices of variables that should be reset. Formally *reset*$(\delta(r), \delta(\lambda))$ is defined as $\delta(r)[\delta(\lambda) \leftarrow 0]$.

Next, the function T_r takes a region r and an expression of grammar ϕ_t and translates it to a `satisfies` function applied to region r that yields an equivalent result as we will show later on.

$$\begin{aligned}
T_r(r, \vec{p}_i + t < c) &= \texttt{satisfies}(r, \vec{p}_i < c)\\
T_r(r, \vec{p}_i + t \leq c) &= \texttt{satisfies}(r, \vec{p}_i \leq c)\\
T_r(r, \vec{p}_i + t > c) &= \texttt{satisfies}(r, \vec{p}_i > c)\\
T_r(r, \vec{p}_i + t \geq c) &= \texttt{satisfies}(r, \vec{p}_i \geq c)\\
T_r(r, f \wedge g) &= T_r(r, f) \wedge T_r(r, g)\\
T_r(r, f \vee g) &= T_r(r, f) \vee T_r(r, g)
\end{aligned}$$

Where f and g are formulae of grammar ϕ_t. Every region r is a region over a real-valued list variable $\vec{u}$. In the next definition a subscript i is used to denote a specific expression of the corresponding grammar. The function REGION is used to translate each predicate formula and is defined as:

$$\begin{aligned}
&\text{REGION}(\exists_{e_i:E_i}.\exists_{t:\text{Real}}.(b \wedge \phi_{t_i}(\vec{p}, t) \wedge X_i(g_i(e_i, d), h_i(t, \vec{p}, \lambda))), \vec{p}, d, r) =\\
&\exists_{e_i:E_i}.\exists_{r':\text{Region}}.(b \wedge \texttt{is_successor}(r', r) \wedge T_r(r', \phi_{t_i}) \wedge \hat{X}_i(g_i(e_i, d), \texttt{reset}(r', \lambda)))\\
&\text{REGION}(\forall_{e_i:E_i}.\forall_{t:\text{Real}}.((b \wedge \phi_{t_i}(\vec{p}, t)) \implies X_i(g_i(e_i, d), h_i(t, \vec{u}, \lambda)), \vec{p}, d, r) =\\
&\forall_{e_i:E_i}.\forall_{r':\text{Region}}.((b \wedge \texttt{is_successor}(r', r) \wedge T_r(r', \phi_{t_i})) \implies \hat{X}_i(g_i(e_i, d), \texttt{reset}(r', \lambda)))
\end{aligned}$$

Where b is a boolean function over bound variable e_i and parameter d and ϕ_t is a time-elapse constraint over $\vec{p}$ using the bound variable t. Finally, the *region equation system* $\hat{\mathcal{E}}$ is obtained by applying function T_R to $\mathcal{E}$, where T_R is defined as:

$$
\begin{aligned}
T_R(\varepsilon) &= \varepsilon \\
T_R((\sigma X(d : D, \vec{u} : \text{List(Real)}) = \bigvee_{i \in I} \phi_{R_i}^{\vee})\mathcal{E}') &= \\
(\sigma \hat{X}(d : D, r : \text{Region}) &= \bigvee_{i \in I} \text{REGION}(\phi_{R_i}^{\vee}, \vec{u}, d, r))T_R(\mathcal{E}') \\
T_R((\sigma X(d : D, \vec{u} : \text{List(Real)}) = \bigwedge_{i \in I} \phi_{R_i}^{\wedge})\mathcal{E}') &= \\
(\sigma \hat{X}(d : D, r : \text{Region}) &= \bigwedge_{i \in I} \text{REGION}(\phi_{R_i}^{\wedge}, \vec{u}, d, r))T_R(\mathcal{E}')
\end{aligned}
$$

Note that for the satisfies function applied in function T_r that r' is a time-successor of r and thus the increment by t no longer occurs in the timing constraint. This translation can now be applied to the real-valued equation system of Example 4.

Example 9. Consider the real-valued equation system from Example 4 again, shown below:

$$
\begin{aligned}
\mu X(d : \text{Nat}, x : \text{Real}, y : \text{Real}) = d \approx 1 \wedge \\
(\exists_{t:\text{Real}}.(2 < x + t \leq 5 \wedge X(1, 0, y + t)) \\
\vee \exists_{t:\text{Real}}.(2 < x + t \leq 5 \wedge y + t \geq 10))
\end{aligned}
$$

To be able to apply the translation it must satisfy all the restrictions defined in Definition 35. For this purpose the real-valued variables must be grouped into a single parameter $\vec{u}$. So, let $\vec{u}$ be a list variable of length two such that $\vec{u}_0$ represents variable x and $\vec{u}_1$ represents variable y. The condition $d \approx 1$ is also pushed into the existential quantification. Furthermore the condition $2 < x + t \leq 5$ is split into $x + t > 2 \wedge x + t \leq 5$ resulting in the equation system:

$$
\begin{aligned}
\mu X(d : \text{Nat}, \vec{u} : \text{List(Real)}) = \exists_{t:\text{Real}}.(d \approx 1 \wedge x + t > 2 \wedge x + t \leq 5 \wedge X(1, [0, y + t])) \\
\vee \exists_{t:\text{Real}}.(d \approx 1 \wedge x + t > 2 \wedge x + t \leq 5 \wedge y + t \geq 10))
\end{aligned}
$$

Now the translation to a region equation system can be applied, resulting in the equation system:

$$
\begin{aligned}
\mu \hat{X}(d : \text{Nat}, r : \text{Region}) = \exists_{r':\text{Region}}.(d \approx 1 \wedge \texttt{is_successor}(r', r) \\
\wedge \texttt{satisfies}(r', x > 2) \wedge \texttt{satisfies}(r', x \leq 5)) \wedge \hat{X}(1, \texttt{reset}(r', \{y\})) \\
\vee \exists_{r':\text{Region}}.(d \approx 1 \wedge \texttt{is_successor}(r', r) \\
\wedge \texttt{satisfies}(r', x > 2) \wedge \texttt{satisfies}(r', x \leq 5) \wedge \texttt{satisfies}(r', y \geq 10))
\end{aligned}
$$

There are 18 equations in the instantiated boolean equation system and the solution for $\hat{X}_{1,[x=y=0]}$ yields true.

This example does not show that the solution for the region equation system is meaningful for the original equation system. For timed automata an equivalence relation was established between all extended states with equal locations and equivalent clock interpretation, i.e. within the same region. For equation systems a similar relation is defined between the solution of the original equation system and the solution of the region equation system. The following theorem states the intended relation.

Theorem 2. *Let $\mathcal{E}$ be an equation system that satisfies the restrictions of Definition 35. For all $X \in bnd(\mathcal{E})$ there exists a region R such that for all values $\vec{v} \in R$ it holds that:*

$$
[\![\mathcal{E}]\!](X)(e, \vec{v}) \iff [\![T_R(\mathcal{E})]\!](\hat{X})(e, R)
$$

This theorem can be proven by showing a specific relation between signatures of both equation systems. This specific relation is called a *consistent correlation*. The theory of such relations and their implications will be introduced in the next section.

5.2 Proof of correctness

For the purpose of proving theorem 2 the theory of a consistent correlation, as presented in [12], is introduced first. A consistent correlation is a relation on signatures that also yields a bi-implication between the solutions of these signatures.

Given a relation on signatures we will define a set of valuations which are *consistent* with that relation. Intuitively this means that the relation is extended to a bi-implication on the signatures in each valuation in the set of consistent valuations.

Definition 37 (Consistent valuations)**.** Let $S \subseteq sig(\mathcal{X}) \times sig(\mathcal{X})$ be a relation on signatures. A valuation η is *consistent* with relation S if $X(v)\ S\ X'(v')$ implies that $\eta(X)(v) \iff \eta(X')(v')$. The set of all valuations consistent with S is denoted by Ω_S.

Next, we can define when a relation is a consistent correlation within a single equation system. For a consistent correlation the ranks of the related predicate variables must be the same. Furthermore the right-hand sides of related predicate variables must have the same solution for its interpretation given a consistent valuation.

Definition 38 (Consistent consequence)**.** Let $\mathcal{E}$ be an equation system. A relation $S \subseteq sig(\mathcal{X}) \times sig(\mathcal{X})$ is a *consistent consequence* on $\mathcal{E}$ if for all $X(v)\ S\ X'(v')$ it holds that:

- $rank_{\mathcal{E}}(X) = rank_{\mathcal{E}}(X')$
- If X and X' are bound in $\mathcal{E}$ then for all $\eta \in \Omega_R$ and all δ we have $[\![\phi_X]\!]\eta\delta[d_X \leftarrow v] \iff [\![\phi_{X'}]\!]\eta\delta[d_{X'} \leftarrow v']$

The consistent correlation is defined on signatures within a single equation system, but the translation results in a different equation system. Two equation systems can be combined while preserving the consistent correlation when these are so-called *compatible*. Two equation systems are compatible whenever predicate variables are not bound in both equation systems. Also, predicate variables occurring in one equation system are not bound by the other equation system and the other way around.

Definition 39 (Compatible)**.** Let $\mathcal{E}$ and $\mathcal{E}'$ be two equation systems, then these are called *compatible* whenever there are no name clashes between the bound variables in both systems and the free variables in one system are not bound by the other. Formally $bnd(\mathcal{E}) \cap bnd(\mathcal{E}') = bnd(\mathcal{E}) \cap occ(\mathcal{E}') = occ(\mathcal{E}) \cap bnd(\mathcal{E}') = \emptyset$.

Then, a combined equation system can be defined of two compatible equation systems that preserves the ranks of all bound predicate variables.

Definition 40. Let $\mathcal{E}$ and $\mathcal{E}'$ be compatible equation systems. A relation $S \subseteq sig(\mathcal{X}) \times sig(\mathcal{X})$ is a consistent consequence between $\mathcal{E}$ and $\mathcal{E}'$ if S is a consistent consequence on some equation system $\mathcal{F}$ consisting of the equations of $\mathcal{E}$ and $\mathcal{E}'$ satisfying:

- $rank_{\mathcal{F}}(X) = rank_{\mathcal{E}}(X)$ for all $X \in bnd(\mathcal{E})$
- $rank_{\mathcal{F}}(X') = rank_{\mathcal{E}'}(X')$ for all $X' \in bnd(\mathcal{E}')$

Finally, the theorem that shows that a consistent correlation between the signatures of two compatible equation systems will induce a bi-implication on their solutions.

Theorem 3. *Let $\mathcal{E}$, $\mathcal{E}'$ be closed, compatible equation systems. Let $S \subseteq sig(\mathcal{X}) \times sig(\mathcal{X})$ be a consistent correlation between $\mathcal{E}$ and $\mathcal{E}'$. We have, for all $X \in bnd(\mathcal{E})$, $X' \in bnd(\mathcal{E}')$:*

$$X(v)\ S\ X'(v') \implies (([\![\mathcal{E}]\!])(X)(v) \iff ([\![\mathcal{E}']\!])(X')(v')) \tag{5.2}$$

Proof. See [12]. □

We will show that the following relation is a consistent correlation between the original equation system and the region equation system.

Definition 41. Let $\mathcal{E}$ be an equation system and $\hat{\mathcal{E}}$ its region equation system. Let $\hat{\sim} \subseteq sig(\mathcal{X}) \times sig(\mathcal{X})$ be a relation such that $X(e, \vec{v}) \hat{\sim} \hat{X}(e, R)$ is defined whenever $\vec{v} \in R$.

First we show that the original equation system and the region equation system are compatible.

Lemma 1. *Let $\mathcal{E}$ be an equation system that satisfies the restrictions of Definition 36, then $\mathcal{E}$ is compatible with $T_R(\mathcal{E})$.*

Proof. In the translation in every equation the bound variable is replaced by a different bound variable. Under the assumption that the replacement variable is not already defined in $\mathcal{E}$ this implies that $bnd(\mathcal{E}) \cap bnd(T_R(\mathcal{E}))$ is empty. The original equation system is closed and as such every occurring variable is also bound, meaning that $occ(\mathcal{E}) \subseteq bnd(\mathcal{E})$. Every occurring predicate variable is replaced by a predicate variable that is bound in the region equation system, this implies that $occ(T_R(\mathcal{E})) \subseteq bnd(T_R(\mathcal{E}))$. These observations imply that $occ(\mathcal{E}) \cap bnd(T_R(\mathcal{E}))$ and $occ(T_R(\mathcal{E})) \cap bnd(\mathcal{E})$ are both empty. This concludes the proof. □

To show a consistent correlation we must also show that the ranks for the related predicate variables are equivalent. We show this for a very general case that applies to both the region and the zone translation that will be defined later on.

Lemma 2. *Let $\mathcal{E}$ be an equation system and let T be a function on equation systems that only replaces predicate variables and predicate formulae, which is defined as:*

$$T(\varepsilon) = \varepsilon$$
$$T((\zeta X(d : D) = \phi_X)\mathcal{E}') = (\zeta X'(d : D) = \phi'_X)T(\mathcal{E}')$$

For every predicate variable $X \in bnd(\mathcal{E})$ it holds that $X' \in bnd(T(\mathcal{E}))$ and $rank_{\mathcal{E}}(X)$ is equal to $rank_{T(\mathcal{E})}(X')$.

Where $\zeta \in \{\mu, \nu\}$.

Proof. Proof by induction on the structure of $\mathcal{E}$.

Base case, $\mathcal{E} \equiv \varepsilon$. There are no bound variables and as such the statement holds.

Induction step, $\mathcal{E} \equiv (\zeta Y(d : D) = \phi_Y)\mathcal{E}'$. The induction hypothesis is that for all $Z \in bnd(\mathcal{E}')$ it holds that $Z' \in bnd(T(\mathcal{E}'))$ and $rank_{\mathcal{E}'}(Z)$ is equal to $rank_{T(\mathcal{E}')}(Z')$. The rank of Y equals one or zero depending on the value of ζ. By applying the function T we obtain $(\zeta Y'(d : D) = \phi'_Y))T(\mathcal{E}')$. As the binding fix point is the same it holds that the rank $rank_{T(\mathcal{E})}(Y')$ is equal to $rank_{\mathcal{E}}(Y)$.

When $\mathcal{E}'$ is equal to ε then the proof statement holds.

When $\mathcal{E}'$ starts with a ζ-block then the ranks for all bound predicate variables of $\mathcal{E}'$ stay the same, i.e. for all $Z \in bnd(\mathcal{E}')$ it holds that $rank_{\mathcal{E}}(Z) = rank_{\mathcal{E}'}(Z)$. From the definition of T we know that $T(\mathcal{E}')$ starts with a ζ-block as well, as such for all $Z' \in bnd(T(\mathcal{E}'))$ it holds that $rank_{T(\mathcal{E})}(Z') = rank_{T(\mathcal{E}')}(Z')$. From the induction hypothesis, for all $Z \in bnd(\mathcal{E}')$ it holds that $rank_{\mathcal{E}'}(Z) = rank_{T(\mathcal{E}')}(Z')$ so the proof statement holds.

When equation system $\mathcal{E}'$ starts with a different block. The ranks for all $Z \in bnd(\mathcal{E}')$ become $rank_{\mathcal{E}}(Z) = rank_{\mathcal{E}'}(Z) + 1$. Again, by definition of T it holds that $T(\mathcal{E}')$ starts with a different block, so for all $Z' \in bnd(T(\mathcal{E}'))$ it holds that $rank_{T(\mathcal{E}')}(Z') = rank_{T(\mathcal{E}')}(Z') + 1$. From the induction hypothesis, for all $Z \in bnd(\mathcal{E}')$ it holds that $rank_{\mathcal{E}'}(Z) = rank_{T(\mathcal{E}')}(Z')$. So the proof statement holds. □

Note that this lemma applies to the region translation T_R as defined in Definition 36 for ϕ'_X given by the application of REGION($\phi^{\wedge}_R, \vec{u}, d, r$) for every conjunct, and similarly for every disjunct, and X' equal to $\hat{X}$.

The next lemma shows an equivalent solution for the interpretation of a timed constraint and the application of the translation function T_r as part of the translation.

Lemma 3. *Let η be a valuation in Val, let δ be a data environment and let R be a region over a real-valued list variable $\vec{u}$ with given upper bounds $c_{\vec{u}}$. Then for all expressions f of grammar ϕ_t, all values $\vec{w} \in \vec{\mathbb{R}}$ and all real numbers $t' \in \mathbb{R}$ the following statement holds.*

$$\vec{v} + t' \in R \implies (\llbracket f \rrbracket \eta\delta[\vec{u} \leftarrow \vec{v}, t \leftarrow t'] \iff \llbracket T_r(r, f) \rrbracket \eta\delta[r \leftarrow R]) \tag{5.3}$$

Proof. Take an arbitrary value $\vec{v} \in \vec{\mathbb{R}}$ and arbitrary value $t' \in \mathbb{R}$. Assume that $\vec{v} + t'$ is an element of the region R. Proof by induction on the structure of f.

Base case $f \equiv \vec{u}_i + t < c$, by calculation:

$$\begin{aligned}
& \llbracket \vec{u}_i + t < c \rrbracket \eta\delta[\vec{u} \leftarrow \vec{v}, t \leftarrow t'] \\
& \quad \{\text{Definition of interpretation applied to b}\} \\
\iff & \delta[\vec{u} \leftarrow \vec{v}, t \leftarrow t'](\vec{u}_i + t < c) \\
& \quad \{\text{Arithmetic}\} \\
\iff & \delta[\vec{u} \leftarrow \vec{v} + t'](\vec{u}_i < c) \\
& \quad \{\text{Definition of satisfies and assumption that } \vec{v} + t' \in R\} \\
\iff & \delta[r \leftarrow R](r \models \vec{u}_i < c) \\
& \quad \{\text{Definition of } \texttt{satisfies}\} \\
\iff & \llbracket \texttt{satisfies}(r, \vec{u}_i < c) \rrbracket \eta\delta[r \leftarrow R] \\
& \quad \{\text{Definition of } T_r\} \\
\iff & \llbracket T_r(r, \vec{u}_i + t < c) \rrbracket \eta\delta[r \leftarrow R]
\end{aligned}$$

Base cases $\vec{u}_i + t \leq c$, $\vec{u}_i + t > c$ and $\vec{u}_i + t \geq c$ are similar.

Case $f \equiv g \wedge h$, where g and h are expressions of grammar ϕ_t. The induction hypothesis is that for both $k \in \{g, h\}$ the interpretation $\llbracket k \rrbracket \eta\delta[\vec{u} \leftarrow \vec{v}, t \leftarrow t'] \iff \llbracket T_r(r, k) \rrbracket \eta\delta[r \leftarrow R]$ holds.

$$\begin{aligned}
& \llbracket g \wedge h \rrbracket \eta\delta[\vec{u} \leftarrow \vec{v}, t \leftarrow t'] \\
& \quad \{\text{Definition of interpretation for conjunction}\} \\
\iff & \llbracket g \rrbracket \eta\delta[\vec{u} \leftarrow \vec{v}, t \leftarrow t'] \wedge \llbracket h \rrbracket \eta\delta[\vec{u} \leftarrow \vec{v}, t \leftarrow t'] \\
& \quad \{\text{Induction hypothesis applied twice}\} \\
\iff & \llbracket T_r(r, g) \rrbracket \eta\delta[r \leftarrow R] \wedge \llbracket T_r(r, h) \rrbracket \eta\delta[r \leftarrow R] \\
& \quad \{\text{Definition of interpretation for conjunction}\} \\
\iff & \llbracket T_r(r, g) \wedge T_r(r, h) \rrbracket \eta\delta[r \leftarrow R] \\
& \quad \{\text{Definition of } T_r\} \\
\iff & \llbracket T_r(r, g \wedge h) \rrbracket \eta\delta[r \leftarrow R]
\end{aligned}$$

Case $f \equiv g \vee h$ is similar, because of the definition of disjunction and function T_r. □

Using the previous two lemmas we can establish that the second condition of Definition 38, which is the definition of a consistent correlation, holds for the relation defined in Definition 41. This is captured in Lemma 4.

Lemma 4. *Let $\mathcal{E}$ be an equation system and $\hat{\mathcal{E}} = T_R(\mathcal{E})$ be its corresponding region equation system. Let $\hat{\sim}$ be a relation on $\mathcal{E}$ and $\hat{\mathcal{E}}$ as defined in Definition 41. Then for all $X \in bnd(\mathcal{E})$ and $\hat{X} \in bnd(\hat{\mathcal{E}})$ it holds that for all regions R and all values $\vec{v} \in R$, all valuations $\eta \in \Omega$ and all data environments δ Equation (5.4) holds.*

$$\llbracket \phi_X \rrbracket \eta\delta[d \leftarrow e, \vec{u} \leftarrow \vec{v}] \iff \llbracket REGION(\phi_X, \vec{u}, d, r) \rrbracket \eta\delta[d \leftarrow e, r \leftarrow R] \tag{5.4}$$

Proof. Take an arbitrary bound variable X, an arbitrary consistent valuation $\eta \in \Omega_{\sim}$ an arbitrary region R and an arbitrary data environment δ. Proof by case distinction on the structure of ϕ_X.

Case $\phi_X \equiv \bigvee_{i\in I} \phi^{\vee}_{R_i}$. This interpretation holds if and only if there exists an element $j \in I$ for which the interpretation of $\phi^{\vee}_{R_i}$ holds. So, by calculation:

$$
\begin{aligned}
&[\![\exists_{e_j:E_j}.\exists_{t:\text{Real}}.(b \wedge \phi_{t_j}(\vec{u},t) \wedge X_j(g_j(e_i,d),h_j(t,\vec{p},\lambda)))]\!]\eta\delta[d \leftarrow e, \vec{u} \leftarrow \vec{v}] \\
&\quad \{\text{Definition of interpretation for existence}\} \\
\iff &\exists_{e'_j\in\mathbb{E}_j}\exists_{t'\in\mathbb{R}}[\![b \wedge \phi_{t_j}(\vec{u},t) \wedge X_j(g_j(e_j,d),h_j(t,\vec{u},\lambda))]\!]\eta\delta[d \leftarrow e, \vec{u} \leftarrow \vec{v}, e_j \leftarrow e'_j, t \leftarrow t']
\end{aligned}
$$

$\Longrightarrow$) Existential elimination, pick values $e'_k \in \mathbb{E}_j$ and $q \in \mathbb{R}$ for which this interpretation holds. Let R' be a region such that $\vec{v} + q \in R'$. As such R' is a time-successor of R.
$\Longleftarrow$) Existential introduction where interpretation holds for values $e'_k \in \mathbb{E}_j$ and $q \in \mathbb{R}$. Note that because R' is a time-successor of region R it is possible to pick an element $\vec{v}$ in region R and real value q such that $\vec{v} + q$ lies within R'.

$$
\begin{aligned}
&\quad \{\text{Observations}\} \\
\iff &[\![b \wedge \phi_{t_j}(\vec{u},t) \wedge X_j(g_j(e_j,d),h_j(t,\vec{u},\lambda))]\!]\eta\delta[d \leftarrow e, \vec{u} \leftarrow \vec{v}, e_j \leftarrow e'_k, t \leftarrow q] \\
&\quad \{\text{Definition of } \texttt{is_successor} \text{ and nothing depends on r or r'}\} \\
\iff &[\![b \wedge \texttt{is_successor}(r',r) \wedge \phi_{t_j}(\vec{u},t) \wedge X_j(g_j(e_j,d),h_j(t,\vec{u},\lambda))]\!] \\
&\quad \eta\delta[d \leftarrow e, \vec{u} \leftarrow \vec{v}, e_j \leftarrow e'_k, t \leftarrow q, r \leftarrow R, r' \leftarrow R'] \\
&\quad \{\text{Lemma 3 with } \vec{v}+q \in R'\} \\
\iff &[\![b \wedge \texttt{is_successor}(r',r) \wedge T_r(r',\phi_{t_j}) \wedge X_j(g_j(e_j,d),h_j(t,\vec{u},\lambda))]\!] \\
&\quad \eta\delta[d \leftarrow e, \vec{u} \leftarrow \vec{v}, e_j \leftarrow e'_k, t \leftarrow q, r \leftarrow R, r' \leftarrow R']
\end{aligned}
$$

By definition of h_j the application $h_j(q,\vec{v},\lambda)$ yields a list $\vec{w}$ such that for all values $0 \le i < |\vec{w}|$ the element in the list $\vec{w}_i$ is equal to zero if $i \in \lambda$ and $\vec{w}_i$ is equal to $\vec{v}+q$ otherwise. Similarly vector $\vec{v}+q$ lies within R', by definition of a time-successor, and by definition of a reset the list $\vec{w}$ is also an element of $R'[\lambda \leftarrow 0]$. As such $X_j(g_j(e_j,d),h_j(t,\vec{u},\lambda)) \sim \hat{X}_j(g_j(e_i,d),\texttt{reset}(r',\lambda)$. So by the definition of a consistent valuation it holds that:

$$
\begin{aligned}
&\quad \{\text{Definition of consistent valuation } \eta\} \\
\iff &[\![b \wedge \texttt{is_successor}(r',r) \wedge T_r(r',\phi_{t_j}) \wedge \hat{X}_j(g_j(e_j,d),\texttt{reset}(r',\lambda)]\!] \\
&\quad \eta\delta[d \leftarrow e, \vec{u} \leftarrow \vec{v}, e_j \leftarrow e'_k, t \leftarrow q, r \leftarrow R, r' \leftarrow R'] \\
&\quad \{\text{The value for } t \text{ is not important}\} \\
\iff &[\![b \wedge \texttt{is_successor}(r',r) \wedge T_r(r',\phi_{t_j}) \wedge \hat{X}_j(g_j(e_j,d),\texttt{reset}(r',\lambda)]\!] \\
&\quad \eta\delta[d \leftarrow e, \vec{u} \leftarrow \vec{v}, e_j \leftarrow e'_k, r \leftarrow R, r' \leftarrow R']
\end{aligned}
$$

$\Longrightarrow$) Finally, existential introduction for the observation of e'_k and R'.
$\Longleftarrow$) Existential elimination taking values $e'_k \in \mathbb{E}_j$ and R' such that the interpretation holds. Observe that R' is a time-successor of R by definition of *is_successor* and definition of conjunction.

$$
\begin{aligned}
&\quad \{\text{Observations}\} \\
\iff &\exists_{e_j\in\mathbb{E}_j}.\exists_{r'\in Region}.[\![b \wedge \texttt{is_successor}(r',r) \wedge \\
&\quad T_r(r',\phi_{t_j}) \wedge \hat{X}_j(g_j(e_j,d),\texttt{reset}(r',\lambda)))]\!]\eta\delta[d \leftarrow e, e_j \leftarrow e'_j, r \leftarrow R, r' \leftarrow R'] \\
&\quad \{\text{Definition of interpretation for existential quantification}\} \\
\iff &[\![\exists_{e_j:E_j}.\exists_{r':\text{Region}}.(b \wedge \texttt{is_successor}(r',r) \\
&\quad \wedge T_r(r',\phi_{t_j}) \wedge \hat{X}_j(g_j(e_j,d),\texttt{reset}(r',\lambda)))]\!]\eta\delta[d \leftarrow e, r \leftarrow R] \\
&\quad \{\text{Definition of REGION}\} \\
\iff &[\![\text{REGION}(\phi^{\vee}_{R_j},\vec{u},d,r)]\!]\eta\delta[d \leftarrow e, r \leftarrow R]
\end{aligned}
$$

Again, this interpretation holds for some $j \in I$ if and only the interpretation of $\bigvee_{i \in I} \text{REGION}(\phi^{\vee}_{R_i}, \vec{u})$ holds as well.

Case $\phi_X \equiv \bigwedge_{i \in I} f_i$, where $f_i \in \phi^{\wedge}_{R}$. Similar, the main observation is that for any t we can choose a time-successor region R' such that for $\vec{v} + t \in R'$ and then lemma 3 can be applied for all R' and t respectively, because of the property of satisfies. □

Finally, it is shown that $\hat{\sim}$ is a consistent correlation between the original equation system and its corresponding region equation system.

Lemma 5. *Let $\mathcal{E}$ be an equation system and $\hat{\mathcal{E}} = T_R(\mathcal{E})$ be its corresponding region equation system. Let $\hat{\sim}$ be a relation on $\mathcal{E}$ and $\hat{\mathcal{E}}$ as defined in 41. The relation $\hat{\sim}$ is a consistent correlation between signatures in equation systems $\mathcal{E}$ and $\hat{\mathcal{E}}$.*

Proof. Let $\mathcal{F}$ be an equation system containing all predicate equations of both $\mathcal{E}$ and $\hat{\mathcal{E}}$ such that the ranks are preserved as defined in Definition 40. The equation systems $\mathcal{E}$ and $\hat{\mathcal{E}}$ are compatible by Lemma 1.

First we show that relation $\hat{\sim}$ is a consistent consequence in $\mathcal{F}$ as defined in Definition 38. This requires showing that for all defined $X(e, \vec{v}) \hat{\sim} \hat{X}(e, R)$ it holds that $rank_F(X) = rank_F(\hat{X})$, which is the same as showing that $rank_{\mathcal{E}}(X) = rank_{\hat{\mathcal{E}}}(\hat{X})$. This holds by Lemma 2 applied to the region translation. By Lemma 4 and the rank observation the two conditions of 38 are satisfied by relation $\hat{\sim}$. From this we can conclude that relation $\hat{\sim}$ is a consistent correlation between $\mathcal{E}$ and $\hat{\mathcal{E}}$. □

We have established a consistent correlation $\hat{\sim}$ between signatures in equation systems $\mathcal{E}$ and $\hat{\mathcal{E}}$. This allows us to prove the main theorem for the region abstraction.

Theorem 4. *Let $\mathcal{E}$ be an equation system that satisfies the restrictions of Definition 35. For all $X \in bnd(\mathcal{E})$ there exists a region R such that for all values $\vec{v} \in R$ it holds that:*

$$[\![\mathcal{E}]\!](X)(e, \vec{v}) \iff [\![T_R(\mathcal{E})]\!](\hat{X})(e, R)$$

Proof. By Lemma 5 there is a consistent correlation $\hat{\sim}$ between signatures of equation systems $\mathcal{E}$ and $\hat{\mathcal{E}}$ such that for all $\vec{v} \in R$ the relation $X(e, \vec{v}) \hat{\sim} \hat{X}(e, R)$ is defined. Finally, by Theorem 3 we can conclude that $[\![\mathcal{E}]\!](X)(e, \vec{v})$ holds if and only if $[\![\hat{\mathcal{E}}]\!](\hat{X})(e, R)$ does. □

The correctness of Theorem 2 implies that the region equation system $\hat{\mathcal{E}}$ offers a suitable abstraction for the solution of the original equation system $\mathcal{E}$. This allows us to conclude that for the equation system of Example 4 the solution for $X_{1,0,0}$ is equal to true because the solution for its corresponding boolean equation in the region equation system from Example 9 has solution $\hat{X}_{1,[x=y=0]}$ yields true and list $[0, 0]$ is the only element of region $[x = y = 0]$.

Chapter 6

Zone abstraction for equation systems

The region abstraction makes it possible to solve equation systems with real-valued parameters provided that the other parameters can also be instantiated to a finite boolean equation system. However, as shown in Example 9 it can result in a large number of regions being considered for the instantiation to a boolean equation system.

One way to reduce the number of regions considered is to combine them. The union of the following regions $[x = 0; 2 < y < 3]$, $[x = 0; y = 3]$, $[x = 0; 3 < y < 4]$, $[x = 0; y = 4]$, $[x = 0; 4 < y < 5]$ and $[x = 0; y = 5]$ introduced in example 9 defines a set of timed data environments, or equivalently a set of lists, where the values assigned to x and y are the same as the values in $\{(x, y) \mid (x, y) \in \vec{\mathbb{R}} \wedge x = 0 \wedge 2 < y \leq 5\}$. Such a convex union of regions is called a *zone*. More generally, a zone is defined as a convex set of real-valued vectors.

A zone can be represented by a conjunction of several difference constraints.

Definition 42 (Representation of zones)**.** The sort Zone over a vector $\vec{u}$ of real-sorted variables represents zones as conjunctive formula of atomic difference constraints:

$$f_z \ ::= \ f_z \wedge f_z \ \mid \ x - y < c \ \mid \ x - y \leq c \ \mid \ x < c \ \mid \ x \leq c \ \mid \ x > c \ \mid \ x \geq c$$

Where x and y are variables in $\vec{u}$ and c is a natural number constant.

A zone describes a set of values that satisfy its constraints. As such the zone $\{(x, y) \mid (x, y) \in \vec{\mathbb{R}} \wedge x = 0 \wedge 2 < y \leq 5\}$ could be represented by the expression $[x \leq, x \leq 0, y \geq 2, y \leq 5]$. Note that the number of zone representations is not bounded. We will see later on how the notion of upper bounds, similar to the upper bounds for regions, can be used to obtain a finite partitioning of the real-valued domain.

Remark 6. The definition of a zone does allow diagonal constraints, i.e., constraints of the form $x - y < c$, where x, y are real-valued variables. However, for the restrictions on equation systems that will be defined, these type of constraints do not occur as part of the equation system.

Note that diagonal constraints are important to describe the behaviour of the fractional ordering in the combined regions. For example, the region $[0 < x < y < 1]$ can be written as a zone with the diagonal constraint $x - y < 0$, which is equal to $x < y$.

Also note that by combining regions into a zone the fact that all values in the zone satisfy the same timing constraints is lost. Consider the example zone $[x > 2, x - y \leq 0, y - x \leq 0, y < 5]$ and the timing constraint $x > 3$. It is no longer true that either every value in this zone satisfies this constraint or no value in this zone does. In fact there is a zone $[x > 2, x - y \leq 0, y - x \leq 0, y \leq 3]$ where for every value this constraint does not hold and a zone $[x > 3, x - y \leq 0, y - x \leq 0, y < 5]$ where every value does satisfy it. This observation implies that for a boolean equation with a given zone it is no longer possible to infer that a constraint either holds for all values in a zone

or for none. As such this is reduced to the notion that there exists an element in that zone that satisfies the predicate formula. This leads to extra restrictions, compared to the equation systems for which the region translation is defined, on the set of equation systems that can be translated using zones.

Again, similar to regions several functions must be defined to progress time and reset variables. Given a zone Z over a real-valued list variable $\vec{u}$ of real-sorted variables the following functions are defined:

- The time-elapse function, denoted by $\Uparrow Z$, results in a zone where every variable can grow unbounded. Formally, the result is the zone $\{\vec{v}+t \mid t \in \mathbb{R}, \vec{v} \in Z\}$. The same symbol as for the set of regions is used, because this represents the same notion.
- The reset function, denoted by $Z[\lambda \leftarrow 0]$, where $\lambda \subseteq 2^{\mathbb{N}}$ is a set of indices, results in a zone where every variable in λ can only be zero, but every other variable can have the same values as it could in zone Z. Formally, this results in the zone $\{\vec{v}[\lambda \leftarrow 0] \mid \vec{v} \in Z\}$. The notation $\vec{v}[S \leftarrow 0]$, where S is a set of indices, is an extension of a function update to sets, formally $\vec{v}[S \leftarrow 0]$ where $S \subseteq 2^{\mathbb{N}}$ results in a list where $\vec{v}_i = 0$ if $i \in S$ and $\vec{v}_i = \vec{v}_i$ otherwise.
- The conjunction function, denoted by $Z \wedge f$ where f is an expression of grammar b_u, results in a zone with the added timing condition. Formally, the result is the zone $\{\vec{v} \mid \vec{v} \in Z \wedge f(\vec{v})\}$
- The emptiness function, denoted by $Z \neq \emptyset$, returns false whenever the zone Z is an empty set and true otherwise.

6.1 Translation

The set of equation systems that can be abstracted using the later defined zone translation is given by the following definition.

Definition 43 (Restrictions for zone translation input equation systems)**.** Let $\phi_t^\wedge$ be the grammar for timing constraints over zones, defined as:

$$\phi_t^\wedge ::= \bigwedge_{j \in J} b_{t_j}$$

Here, b_t is a timing constraint as defined in Definition 34 over variables in $\vec{u}$ and a free variable t. Let $\phi_{Z_i}^\vee$ and $\phi_{Z_i}^\wedge$ be the grammar for each predicate equation, defined as:

$$\begin{aligned} \phi_Z^\vee &::= \exists_{e_i:E_i}.\exists_{t:\text{Real}}.(b \wedge \phi_t^\wedge(t) \wedge X_i(g_i(e_i,d), h_j(t,\vec{u},\lambda)) \\ \phi_Z^\wedge &::= \forall_{e_i:E_i}.\forall_{t:\text{Real}}.(\neg b \vee \neg\phi_t^\wedge(t)) \end{aligned}$$

Note that the universal quantification is only used to show the relation with the input equation system for the region translation. The definition of $h_i(t,\vec{u},\lambda)$ is exactly the same as defined in Definition 35.

Finally, the input equation $\mathcal{E}$ is a closed equation system with predicate equations $X(d : D, \vec{u} : \vec{\text{Real}}) = \phi_Z$ where X is a predicate variable in $\mathcal{X}$ and ϕ_Z is given by the following grammar:

$$\phi_Z ::= \bigvee_{i \in I} \phi_Z^\vee(i) \mid \bigwedge_{i \in I} \phi_Z^\wedge(i)$$

Now we can define a *zone equation system* that uses zones to abstract away from real-valued variables.

Definition 44. Let $\mathcal{E}$ be a closed equation system given by the grammar in Definition 43. For the functions on zones the following syntax is introduced:

- The function `time_elapse` : Zone $\rightarrow$ Zone takes a zone z and applies the time-elapse function to it. Formally, $\mathit{time_elapse}(\delta(z))$ equals $\Uparrow \delta(z)$.
- The function `reset` : Zone $\times\ 2^{\mathbb{N}} \rightarrow$ Zone takes a zone z and a set of natural numbers λ and applies the reset function. Formally, $\mathit{reset}(\delta(z), \delta(\lambda))$ equals $\delta(z)[\delta(\lambda) \leftarrow 0]$.
- The function `conjunction` : Zone $\times$ Expression $\rightarrow$ Zone takes a zone z and an expression e of grammar b_u. This operation returns a zone with the timing condition e added. Formally $\mathit{conjunction}(\delta(z), \delta(e))$ is defined as $\delta(z) \wedge \delta(e)$.
- The function `is_notempty` : Zone $\rightarrow$ Bool takes a zone z and returns true if there is at least one value that satisfies this zone and returns false otherwise. Formally $\mathit{is_notempty}(\delta(z))$ is defined as $\delta(z) \neq \emptyset$.
- The function `is_empty` : Zone $\rightarrow$ Bool is the inverse of *is_notempty*. Formally $\mathit{is_empty}(\delta(z))$ is defined as $\neg \mathit{is_notempty}(\delta(z))$.

Similar to regions the timing constraint is changed to a constraint on $\vec{u}$, because the increment by t is part of the time-elapse. This is captured by the following function:

$$\begin{aligned} T_t(\vec{p_i} + t < c) &= \vec{p_i} < c \\ T_t(\vec{p_i} + t \leq c) &= \vec{p_i} \leq c \\ T_t(\vec{p_i} + t > c) &= \vec{p_i} > c \\ T_t(\vec{p_i} + t \geq c) &= \vec{p_i} \geq c \end{aligned}$$

Let T_z be a function that takes a zone and a formula of grammar $\phi_t^{\wedge}$ and returns a zone with the timing constraints added. It is defined as followed:

$$\begin{aligned} T_z(z, \bigwedge_{j \in \emptyset} b_{t_j}) &= \texttt{time_elapse}(z) \\ T_z(z, \bigwedge_{j \in J' \cup \{k\}} b_{t_j}) &= \texttt{conjunction}(T_z(z, \bigwedge_{j \in J'} b_{t_j}), T_t(b_{t_j})) \end{aligned}$$

Let ZONE be a function that translates each predicate formula:

$$\begin{aligned} &\text{ZONE}(\exists_{e_i:E_i}.\exists_{t:\text{Real}}.(b \wedge \phi_{t_i}^{\wedge}(\vec{p}, t) \wedge X_i(g_i(e_i, d), h_i(t, \vec{p}, \lambda))), \vec{p}, d, z) = \\ &\qquad \exists_{e_i:E_i}.(b \wedge \texttt{is_notempty}(T_z(z, \phi_{t_i}^{\wedge})) \wedge X_i(g_i(e_i, d), \texttt{reset}(T_z(z, \phi_{t_i}^{\wedge}), \lambda))) \\ &\text{ZONE}(\forall_{e_i:E_i}.\forall_{t:\text{Real}}.((\neg b \vee \neg \phi_{t_i}^{\wedge}(\vec{p}, t))), \vec{p}, d, z) = \\ &\qquad \forall_{e_i:E_i}.(\neg b) \vee \texttt{is_empty}(T_z(z, \phi_{t_i}^{\wedge})) \end{aligned}$$

The *zone equation system* $\tilde{\mathcal{E}}$ is then obtained by applying T_Z to $\mathcal{E}$, where T_Z is defined as:

$$\begin{aligned} T_Z(\varepsilon) &= \varepsilon \\ T_Z((X(d:D, \vec{u}:\text{List(Real)}) \stackrel{\nu}{=} \bigvee_{i \in I} \phi_{Z_i}^{\vee})\mathcal{E}') &= (\tilde{X}(d:D, z:\text{Zone}) \stackrel{\nu}{=} \bigvee_{i \in I} \text{ZONE}(\phi_{Z_i}^{\vee}, \vec{u}, d, z))T_Z(\mathcal{E}') \\ T_Z((X(d:D, \vec{u}:\text{List(Real)}) \stackrel{\nu}{=} \bigwedge_{i \in I} \phi_{Z_i}^{\wedge})\mathcal{E}') &= (\tilde{X}(d:D, z:\text{Zone}) \stackrel{\nu}{=} \bigwedge_{i \in I} \text{ZONE}(\phi_{Z_i}^{\wedge}, \vec{u}, d, z))T_Z(\mathcal{E}') \end{aligned}$$

This translation can now be applied to the initial example equation system with real-valued variables. This requires the same changes as were made before the region translation could be applied.

Example 10. Consider the real-valued equation system from Example 9, which is the real-valued variant of the initial example with some changes applied to be able to apply the translation, shown below:

$$\begin{aligned}\mu X(d:\text{Nat},\vec{u}:\text{List(Real)}) &= \exists_{t:\text{Real}}.(d\approx 1\wedge x+t>2\wedge x+t\leq 5\wedge X(1,[0,y+t]))\\ &\vee \exists_{t:\text{Real}}.(d\approx 1\wedge x+t>2\wedge x+t\leq 5\wedge y+t\geq 10))\end{aligned}$$

This equation system can then be translated using the zone translation, resulting in:

$$\begin{aligned}&\mu\tilde{X}(d:\text{Nat},z:\text{Zone}) = (\texttt{is_notempty}(Z_0)\wedge\tilde{X}(1,\texttt{reset}(Z_0,\{0\})))\vee\texttt{is_notempty}(Z_1)\\ &Z_0 := \texttt{conjunction}(\texttt{conjunction}(\texttt{time_elapse}(z),x>2),x\leq 5)\\ &Z_1 := \texttt{conjunction}(\texttt{conjunction}(\texttt{conjunction}(\texttt{time_elapse}(z),x>2),x\leq 5),y\geq 10)\end{aligned}$$

The auxiliary variables Z_0 and Z_1 are only used for clarity. Note that this equation system can only be instantiated automatically by the tool-set when it is able to prove that when the zone given by Z_1 is not empty the value for Z_0 no longer matters, preventing it from instantiating ever increasing zones. Later on in the implementation part we will see that upper bounds can be used, in a similar way as for regions, to limit the number of zones that have to be considered.

Solving this equation system only required three boolean equations to be instantiated and the solution for $\tilde{X}_{1,Z}$, where Z is the zone only containing $(0,0)$ given by a conjunction $[x-0\leq 0\wedge 0-x\leq 0\wedge y-0\leq 0\wedge 0-y\leq 0]$.

Similar to regions, a bi-implication between the solution of the original equation system and the zoned equation system is shown. This is captured by the following theorem.

Theorem 5. *Let $\mathcal{E}$ be an equation systems that satisfies the restrictions of Definition 43. For all predicate variables $X\in bnd(\mathcal{E})$ there is a zone Z such that the following equation holds:*

$$\exists_{\vec{v}\in Z}([\![\mathcal{E}]\!](X)(e,\vec{v})) \iff [\![T_Z(\mathcal{E})]\!](\tilde{X})(e,Z) \tag{6.1}$$

We will prove Theorem 5 by a proof graph construction. Given the proof graph for equation system $\mathcal{E}$ we show that it is possible to construct a proof graph for $T_Z(\mathcal{E})$. Likewise, given a proof graph for $T_Z(\mathcal{E})$ the proof graph for $\mathcal{E}$ can be constructed. As these proof graphs are related to the solution of their corresponding equation systems, these observations suffice to show the bi-implication on the solutions and thus proving the theorem.

6.2 Proof of correctness

The next section shows the proof graph construction for the original equation system given the proof graph for the zoned equation. However, we will first prove two lemmas that apply to both constructions. First, Lemma 6 relates the conjunction of a number of timing constraints to the zone resulting from applying T_z.

Lemma 6. *Let η be a valuation in Val and let δ be a data environment. Then for all expressions f of grammar $\phi_t^{\wedge}$ and for all zones Z over real-valued list variable $\vec{u}$ he following statement holds:*

$$\forall_{\vec{v}\in Z}\forall_{t'\in\mathbb{R}}([\![f]\!]\eta\delta[\vec{u}\leftarrow\vec{v},t\leftarrow t'] \iff \vec{v}+t'\in T_z(Z,f))$$

Proof. Proof by structural induction on f. Take an arbitrary zone Z, an arbitrary element $\vec{v}\in Z$ and an arbitrary real number $t'\in\mathbb{R}$.

Case $f \equiv \bigwedge_{j\in\emptyset} b_{t_j}$, applying $T_z(Z, \bigwedge_{j\in\emptyset} b_{t_j})$ yields $\texttt{time_elapse}(Z)$ by definition. By calculation:

$$
\begin{aligned}
& [\![\bigwedge_{j\in\emptyset} b_{t_j}]\!]\eta\delta[\vec{u} \leftarrow \vec{v}, t \leftarrow t'] \\
& \quad \{\text{Definition of interpretation for conjunction}\} \\
\iff & \bigwedge_{j\in\emptyset} [\![b_{t_j}]\!]\eta\delta[\vec{u} \leftarrow \vec{v}, t \leftarrow t'] \\
& \quad \{\text{Definition of conjunction over empty set}\} \\
\iff & \text{true} \\
& \quad \{\text{Definition of set notation and } \vec{v} \in Z\} \\
\iff & \vec{v} + t' \in \{\vec{u} + t \mid \vec{u} \in Z, t \in \mathbb{R}\} \\
& \quad \{\text{Definition of } \texttt{time_elapse}\} \\
\iff & \vec{v} + t' \in \texttt{time_elapse}(Z) \\
& \quad \{\text{Definition of translation}\} \\
\iff & \vec{v} + t' \in T_z(Z, \bigwedge_{j\in\emptyset} b_{t_j})
\end{aligned}
$$

Case $f \equiv \bigwedge_{j\in J} b_{t_j}$. The induction hypothesis states that for any J' the following statement holds:

$$
\forall_{\vec{v}\in Z}\forall_{t'\in\mathbb{R}}([\![\bigwedge_{j\in J'} b_{t_j}]\!]\eta\delta[\vec{u} \leftarrow \vec{v}, t \leftarrow t'] \iff \vec{v} + t' \in T_z(z, \bigwedge_{j\in J'} b_{t_j}))
$$

Case $f \equiv \bigwedge_{j\in J'\cup\{k\}} b_{t_j}$:

$$
\begin{aligned}
& [\![\bigwedge_{j\in J'\cup\{k\}} b_{t_j}]\!]\eta\delta[\vec{u} \leftarrow \vec{v}, t \leftarrow t'] \\
& \quad \{\text{Definition of conjunction }\} \\
\iff & [\![b_{t_k} \wedge \bigwedge_{j\in J'} b_{t_j}]\!]\eta\delta[\vec{u} \leftarrow \vec{v}, t \leftarrow t'] \\
& \quad \{\text{Definition of interpretation for conjunction }\} \\
\iff & [\![b_{t_k}]\!]\eta\delta[\vec{u} \leftarrow \vec{v}, t \leftarrow t'] \wedge [\![\bigwedge_{j\in J'} b_{t_j}]\!]\eta\delta[\vec{u} \leftarrow \vec{v}, t \leftarrow t'] \\
& \quad \{\text{Definition of } T_t \text{ and induction hypothesis}\} \\
\iff & [\![T_t(b_{t_k})]\!]\eta\delta[\vec{u} \leftarrow (\vec{v} + t')] \wedge (\vec{v} + t') \in T_z(z, \bigwedge_{j\in J'} b_{t_j}) \\
& \quad \{\text{Definition of conjunction}\} \\
\iff & \vec{v} + t' \in \texttt{conjunction}(T_z(Z, \bigwedge_{j\in J'} b_{t_j}), T_t(b_{t_k})) \\
& \quad \{\text{Definition of } T_z\} \\
& \vec{v} + t' \in T_z(Z, \bigwedge_{j\in J'\cup\{k\}} b_{t_j})
\end{aligned}
$$

□

Note that a similar proof can be used to show that the zone $T_z(Z, f)$ is not empty if and only if there is an element in $\vec{v} \in Z$ and $t' \in \mathbb{R}$ such that the interpretation of f holds. This is captured in the following lemma:

Lemma 7. *Let η be a valuation in Val and let δ be a data environment. Then for all expressions f of grammar $\phi_t^{\wedge}$ and for all zones Z the following statement holds:*

$$\exists_{\vec{v}\in Z}\exists_{t'\in\mathbb{R}}([\![f]\!]\eta\delta[\vec{u}\leftarrow\vec{v},t\leftarrow t']) \iff [\![\mathtt{is_notempty}(T_z(z,f))]\!]\eta\delta[z\leftarrow Z]$$

Proof. $\implies$). There are elements $\vec{v}\in\vec{\mathbb{R}}$ and $t'\in\mathbb{R}$ such that $[\![f]\!]\eta\delta[\vec{u}\leftarrow\vec{v},t\leftarrow t']$ holds, as such by applying Lemma 6 the value $\vec{v}+t'$ is an element of $T_z(Z,f)$, which by definition of `is_notempty` implies that the interpretation of $\mathtt{is_notempty}(T_z(z,f))$ holds as well.

$\implies$). This case is slightly more involved, but the main argument is that there are values $\vec{v}\in Z$ and $t'\in\mathbb{R}$ such that $\vec{v}+t'$ is an element of $\mathtt{time_elapse}(Z)$ by definition of set resulting from a time-elapse. And that these values satisfy every conjunction which is obtained by a derivation similar to the one used in the previous proof. This means that the value of $\vec{v}+t'$ is an element of $T_z(Z,f)$. After which Lemma 6 can be applied to obtain that for these values for $\vec{v}$ and t' the interpretation of f holds. So by existential introduction on these values the existence of $\vec{v}$ and t' can be obtained. □

Lemma 8 relates to vertices in a graph without any successors, these are called *sinks*. We show that whenever there is sink in a proof graph of the original equation system there can also be sink added to the proof graph of the zoned equation system, and the other way around.

Lemma 8. *Let $\mathcal{E}$ be an equation system that satisfies the restrictions defined in Definition 43. Let $\tilde{\mathcal{E}}$ be its corresponding zoned equation system, given by $T_Z(\mathcal{E})$. For all bound predicate variables $X\in bnd(\mathcal{E})$, for all zones Z there is an element $\vec{v}\in Z$ such that equation 6.2 holds.*

$$[\![\phi_X]\!]\eta[sig(\mathcal{E})\mapsto \mathit{false}]\delta[d\leftarrow e,\vec{u}\leftarrow\vec{v}] \iff [\![\phi_{\tilde{X}}]\!]\eta[sig(\tilde{\mathcal{E}})\mapsto \mathit{false}]\delta[d\leftarrow e,z\leftarrow Z] \quad (6.2)$$

Proof. Proof by case distinction on the structure of ϕ_X.

Case $\phi_X\equiv\bigvee_{i\in I}\phi_{Z_i}^{\vee}$. Take an arbitrary $\phi_{Z_j}^{\vee}$, by calculation:

$$\begin{aligned}
&[\![\exists_{e_j:E_j}.\exists_{t:\mathrm{Real}}.(b\wedge\phi_{t_j}^{\wedge}(\vec{u},t)\wedge X_j(g_j(e_j,d),h_j(t,\vec{u},\lambda))]\!]\eta[sig(\mathcal{E})\mapsto\mathrm{false}]\delta[d\leftarrow e,\vec{u}\leftarrow\vec{v}]\\
&\qquad\{\text{Definition of interpretation for existential quantification}\}\\
\iff&\exists_{e_j\in\mathbb{E}_j}\exists_{t'\in\mathbb{R}}[\![(b\wedge\phi_{t_j}^{\wedge}(\vec{u},t)\wedge X_j(g_j(e_j,d),h_j(t,\vec{u},\lambda))]\!]\eta[sig(\mathcal{E})\mapsto\mathrm{false}]\\
&\qquad\delta[d\leftarrow e,\vec{u}\leftarrow\vec{v},e_j\leftarrow e_j',t\leftarrow t']\\
&\qquad\{\text{Definition of interpretation for conjunction}\}\\
\iff&\exists_{e_j\in\mathbb{E}_j}\exists_{t'\in\mathbb{R}}(...\wedge[\![X_j(g_j(e_j,d),h_j(t,\vec{u},\lambda))]\!]\eta[sig(\mathcal{E})\mapsto\mathrm{false}]\delta[d\leftarrow e,\vec{u}\leftarrow\vec{v},e_j\leftarrow e_j',t\leftarrow t'])\\
&\qquad\{\text{Definition of interpretation for predicate variables}\}\\
\iff&\exists_{e_j\in\mathbb{E}_j}\exists_{t'\in\mathbb{R}}(...\wedge\eta[sig(\mathcal{E})\mapsto\mathrm{false}](X_j)(\delta(g_j(e_j,d)),\delta(h_j(t,\vec{u},\lambda))))\\
&\qquad\{\text{Definition of function update }\}\\
\iff&\mathrm{false}
\end{aligned}$$

Similarly, after applying the zone translation T_Z. Take an arbitrary $\phi_{Z_j}^{\vee}$ and apply the ZONE

function ZONE($\phi^{\vee}_{t_j}, \vec{u}, d, z$). By calculation:

$$
\begin{aligned}
& [\![\exists_{e_j:E_j}.(b \wedge \texttt{is_notempty}(T_z(z, \phi^{\wedge}_{t_j})) \wedge \tilde{X}_j(g_j(e_j, d), \texttt{reset}(T_z(z, f_t), \lambda)))]\!] \\
& \quad \eta[sig(\tilde{\mathcal{E}}) \mapsto \text{false}]\delta[d \leftarrow e, z \leftarrow Z] \\
& \quad \{\text{Definition of interpretation for existential quantification}\} \\
\iff & \exists_{e'_j \in \mathbb{E}_j} [\![b \wedge \texttt{is_notempty}(T_z(z, \phi^{\wedge}_{t_j})) \wedge \tilde{X}_j(g_j(e_j, d), \texttt{reset}(T_z(z, f_t), \lambda))]\!] \\
& \quad \eta[sig(\tilde{\mathcal{E}}) \mapsto \text{false}]\delta[d \leftarrow e, z \leftarrow Z, e_i \leftarrow e'_i] \\
& \quad \{\text{Definition of interpretation for conjunction}\} \\
\iff & \exists_{e'_i \in \mathbb{E}_i}(... \wedge [\![\tilde{X}_j(g_j(e_j, d), \texttt{reset}(T_z(z, \phi^{\wedge}_{t_j})))]\!]\eta[sig(\tilde{\mathcal{E}}) \mapsto \text{false}]]\delta[d \leftarrow e, z \leftarrow Z, e_j \leftarrow e'_j]) \\
& \quad \{\text{Definition of interpretation for predicate variables}\} \\
\iff & \exists_{e'_i \in \mathbb{E}_i}(... \wedge \eta[sig(\tilde{\mathcal{E}}) \mapsto \text{false}](\tilde{X}_j)(\delta(g_j(e_j, d)), \delta(h_j(t, \vec{u}, \lambda)))) \\
& \quad \{\text{Definition of function update}\} \\
\iff & \text{false}
\end{aligned}
$$

Case $\phi_X \equiv \bigwedge_{i \in I} \phi^{\wedge}_{Z_i}$. Take an arbitrary $\phi^{\wedge}_{Z_j}$. By definition of a proof graph and conjunction the following interpretation must be true. We show how to relate the two interpretations by a calculation:

$$
\begin{aligned}
& [\![\forall_{e_j:E_j} \forall_{t:\text{Real}}(\neg b \vee \neg\phi^{\wedge}_{t_j}(\vec{p}, t))]\!]\eta[sig(\mathcal{E}) \mapsto \text{false}]\delta[d \leftarrow e, \vec{u} \leftarrow \vec{v}] \\
& \quad \{\text{Definition of b does not depend on } t'\} \\
\iff & [\![\forall_{e_j:E_j}.(\neg b \vee \forall_{t:\text{Real}}.(\neg\phi^{\wedge}_{t_j}(\vec{p}, t)))]\!]\eta[sig(\mathcal{E}) \mapsto \text{false}]\delta[d \leftarrow e, \vec{u} \leftarrow \vec{v}] \\
& \quad \{\text{Definition of interpretation for universal quantification}\} \\
\iff & \forall_{e'_j \in \mathbb{E}_j} [\![\neg b \vee \forall_{t:\text{Real}}.(\neg\phi^{\wedge}_{t_j}(\vec{p}, t)))]\!]\eta[sig(\mathcal{E}) \mapsto \text{false}]\delta[d \leftarrow e, \vec{u} \leftarrow \vec{v}, e_j \leftarrow e'_j]
\end{aligned}
$$

Now we show an equivalence for the interpretation of $\neg\phi^{\wedge}_{t_j}(\vec{p}, t)$. Take an arbitrary $e'_k \in \mathbb{E}_j$, the calculation:

$$
\begin{aligned}
& [\![\forall_{t:\text{Real}}.(\neg\phi^{\wedge}_{t_j}(\vec{p}, t)))]\!]\eta[sig(\mathcal{E}) \mapsto \text{false}]\delta[d \leftarrow e, \vec{u} \leftarrow \vec{v}, e_j \leftarrow e'_k] \\
& \quad \{\text{Definition of interpretation for universal quantification})\} \\
\iff & \forall_{t' \in \mathbb{R}} [\![\neg\phi^{\wedge}_{t_j}(\vec{p}, t)]\!]\eta[sig(\mathcal{E}) \mapsto \text{false}]\delta[d \leftarrow e, \vec{u} \leftarrow \vec{v}, e_j \leftarrow e'_k, t \leftarrow t'] \\
& \quad \{\text{Duality of universal and existential quantification (Note that } \phi^{\wedge}_{t_j} \text{ is a boolean formula)}\} \\
\iff & \neg\exists_{t' \in \mathbb{R}} [\![\phi^{\wedge}_{t_j}(\vec{p}, t)]\!]\eta[sig(\mathcal{E}) \mapsto \text{false}]\delta[d \leftarrow e, \vec{u} \leftarrow \vec{v}, e_j \leftarrow e'_j, t \leftarrow t'] \\
& \quad \{\text{Lemma 7 applied to } \texttt{is_notempty}\} \\
\iff & \neg[\![\texttt{is_notempty}(T_z(z, \phi^{\wedge}_{t_j}))]\!]\eta[sig(\mathcal{E}) \mapsto \text{false}]\delta[d \leftarrow e, e_j \leftarrow e'_k, z \leftarrow Z] \\
& \quad \{\text{Definition of } \texttt{is_empty}\} \\
\iff & [\![\texttt{is_empty}(T_z(z, \phi^{\wedge}_{t_j}))]\!]\eta[sig(\mathcal{E}) \mapsto \text{false}]\delta[d \leftarrow e, e_j \leftarrow e'_k, z \leftarrow Z]
\end{aligned}
$$

Now applying this equivalence results in:

$$
\begin{aligned}
& \forall_{e'_j \in \mathbb{E}_j} [\![\neg b \vee \forall_{t:\text{Real}}.(\neg\phi^{\wedge}_{t_j}(\vec{p}, t)))]\!]\eta[sig(\mathcal{E}) \mapsto \text{false}]\delta[d \leftarrow e, \vec{u} \leftarrow \vec{v}, e_j \leftarrow e'_j] \\
& \quad \{\text{Equivalence, and } b \text{ does not depend on } \vec{u} \text{ or } z\} \\
\iff & \forall_{e'_j \in \mathbb{E}_j} [\![\neg b \vee \texttt{is_empty}(T_z(z, \phi^{\wedge}_{t_j}))]\!]\eta[sig(\mathcal{E}) \mapsto \text{false}]\delta[d \leftarrow e, e_j \leftarrow e'_j, z \leftarrow Z] \\
& \quad \{\text{Definition of interpretation for universal quantification}\} \\
\iff & [\![\forall_{e_j:E_j}.(\neg b) \vee \texttt{is_empty}(T_z(z, \phi^{\wedge}_{t_j}))]\!]\eta[sig(\mathcal{E}) \mapsto \text{false}]\delta[d \leftarrow e, z \leftarrow Z] \\
& \quad \{\text{Definition of ZONE, and independent of valuation}\} \\
\iff & [\![\text{ZONE}(\phi^{\wedge}_{Z_i}, \vec{u}, d, z)]\!]\eta[sig(\tilde{\mathcal{E}}) \mapsto \text{false}]\delta[d \leftarrow e, z \leftarrow Z]
\end{aligned}
$$

So in this case the interpretation for $\bigwedge_{i\in I}\phi^{\wedge}_{Z_i}$ holds if and only if $\bigwedge_{i\in I}\text{ZONE}(\phi^{\wedge}_{Z_i},\vec{u})$ does. So Equation (6.2) holds and the proof is concluded. □

Note that Equation (6.2) is the definition of a vertex belonging to a proof graph without any successors. in the next section we prove that given the proof graph for the zoned equation system we can construct a proof graph for the original equation system.

6.2.1 Constructing the original proof graph

First we show that for every edge in the proof graph of the zoned equation system a relation can be found between signatures of the original equation system that satisfy the fulfilment property. The way that the zones are defined make it possible to have a single edge for each vertex in the graph. This is formalized in Lemma 9.

Lemma 9. *Let $\mathcal{E}$ be an equation system that satisfies the restrictions defined in Definition 43. Let $\tilde{\mathcal{E}} = T_Z(\mathcal{E})$ be its corresponding zoned equation system. Let $G = \langle V, \rightarrow\rangle$ be a minimal proof graph for $\tilde{\mathcal{E}}$. For every edge $\tilde{X}(e,Z) \rightarrow \tilde{Y}(f,Z')$ it holds that for every element $\vec{w} \in Z'$ there exists an element $\vec{v} \in Z$ such that two signatures $X(e,\vec{v}), Y(f,\vec{w}) \in sig(\mathcal{E})$ exist where Equation (6.3) holds.*

$$[\![\phi_X]\!]\eta[sig(\mathcal{E}) \mapsto \mathit{false}][\{(Y,f,\vec{w})\} \mapsto \mathit{true}]\delta[d \leftarrow e, \vec{u} \leftarrow \vec{v}] \tag{6.3}$$

Proof. Take an arbitrary edge $\tilde{X}(e,Z) \rightarrow \tilde{Y}(e',Z')$ and an arbitrary element $\vec{w} \in Z'$. Proof by case distinction on the structure of $\phi_{\tilde{X}}$.

Case $\phi_{\tilde{X}} \equiv \bigvee_{i\in I}\text{ZONE}(\phi^{\vee}_{Z_i},\vec{u},d,z)$. From the minimality of G and the definition of disjunction it can be inferred that there exists an $j \in I$ such that the following interpretation yields true, otherwise the proof graph is not minimal and the edge could be removed. Let $\tilde{\eta}$ be a valuation defined as $\eta[sig(\tilde{\mathcal{E}}) \mapsto \text{false}][\{(\tilde{Y},f,Z'\} \mapsto \text{true}]$.

$$
\begin{aligned}
&[\![\exists_{e_j:E_j}.(b \wedge \texttt{is_notempty}(T_z(z,\phi^{\wedge}_{t_j})) \wedge \tilde{Y}(g_j(e_j,d), \texttt{reset}(T_z(z,\phi^{\wedge}_{t_j}),\lambda)))]\!]\tilde{\eta}\delta[d \leftarrow e, z \leftarrow Z] \\
&\quad \{\text{Definition of interpretation for existential quantification}\} \\
\iff &\exists_{e'_j\in\mathbb{E}_j}[\![b \wedge \texttt{is_notempty}(T_z(z,\phi^{\wedge}_{t_i})) \wedge \tilde{Y}(g_j(e_j,d), \texttt{reset}(T_z(z,\phi^{\wedge}_{t_j}),\lambda))]\!] \\
&\quad \tilde{\eta}\delta[d \leftarrow e, z \leftarrow Z, e_j \leftarrow e'_j]
\end{aligned}
$$

Existential elimination, take an element $e'_k \in \mathbb{E}_j$ where this interpretations holds. Let η' be a valuation defined as $\eta[sig(\mathcal{E}) \mapsto \text{false}][\{(Y,f,\vec{w})\} \mapsto \text{true}]$. A calculation for b:

$$
\begin{aligned}
\iff &[\![b]\!]\tilde{\eta}\delta[d \leftarrow e, z \leftarrow Z, e_j \leftarrow e'_k] \\
&\quad \{\text{Definition of b, does not depend on z and valuation}\} \\
\iff &[\![b]\!]\eta'\delta[d \leftarrow e, e_j \leftarrow e'_k] \\
&\quad \{\text{Definition of b, does not depend on t}\} \\
\iff &[\![b]\!]\eta'\delta[d \leftarrow e, e_j \leftarrow e'_k, t \leftarrow q]
\end{aligned} \tag{6.4}
$$

The interpretation of $\texttt{is_notempty}(T_z(Z,\phi^{\wedge}_{t_j}))$ implies that $T_z(Z,\phi^{\wedge}_{t_j})$ is not empty. The `reset` function always returns an element for every element in the given zone by definition, as such $\texttt{reset}(T_z(z,\phi^{\wedge}_{t_j}),\lambda))$ is not empty as well. We know that $g_j(e'_k,e)$ is equal to f and Z' is equal to $\texttt{reset}(T_z(z,\phi^{\wedge}_{t_j}),\lambda))$. Take an element $\vec{w}'$ such that $\vec{w}'[\lambda \leftarrow 0]$ equals $\vec{w}$, this is possible by definition of `reset`. By definition of `time_elapse`, which is applied by T_z, this $\vec{w}'$ is the result of an addition of some $\vec{v} \in Z$ and an element $q \in \mathbb{R}$. By definition of the function h_j and the definition of function reset on λ the result of applying $h_j(\vec{v},q,\lambda)$ is equal to $\vec{w}$ as well. By lemma 6 it holds that for this $\vec{v}$ and a real number q such that $\vec{v}+t'$ lies within $T_z(Z,\phi^{\wedge}_{t_j})$. First a derivation

for the is_notempty part.

$$
\begin{aligned}
&[\![\texttt{is_notempty}(T_z(z, \phi^{\wedge}_{t_i}))]\!]\tilde{\eta}\delta[d \leftarrow e, z \leftarrow Z, e_j \leftarrow e'_k] \\
&\quad \{\text{Lemma 7, pick } \vec{v} \text{ and } q\} \\
\iff &[\![\phi^{\wedge}_{t_j}(\vec{u}, t)]\!]\eta'\delta[d \leftarrow e, \vec{u} \leftarrow \vec{v}, t \leftarrow q] \\
&\quad \{\text{Does not depend on value for } e_j\} \\
\iff &[\![\phi^{\wedge}_{t_j}(\vec{u}, t)]\!]\eta'\delta[d \leftarrow e, \vec{u} \leftarrow \vec{v}, t \leftarrow q, e_j \leftarrow e'_k] \qquad (6.5)
\end{aligned}
$$

The following is true by definition of the valuation, because $g_j(e'_k, e)$ is equal to f and $h_j(q, \vec{v}, \lambda)$ is equal to $\vec{w}$:

$$
\begin{aligned}
&\eta'(Y)(\delta[d \leftarrow e, \vec{u} \leftarrow \vec{v}, t \leftarrow q, e_j \leftarrow e'_k](g_j(e_j, d)), \delta[d \leftarrow e, \vec{u} \leftarrow \vec{v}, t \leftarrow q, e_j \leftarrow e'_k](h_j(t, \vec{u}, \lambda))) \\
&\quad \{\text{Definition of interpretation for predicate variables}\} \\
\iff &[\![X_j(g_j(e_j, d), h_j(t, \vec{u}, \lambda))]\!]\eta'\delta[d \leftarrow e, \vec{u} \leftarrow \vec{v}, e_j \leftarrow e'_j, t \leftarrow q] \qquad (6.6)
\end{aligned}
$$

Combining these observations, by definition of conjunction and existential introduction for e'_k and q this yields:

$$
\begin{aligned}
&\quad \{\text{Observations 6.4, 6.5 and 6.6, and existential introductions}\} \\
\iff &\exists_{e'_j \in \mathbb{E}_j} \exists_{t' \in \mathbb{R}} [\![b \wedge \phi^{\wedge}_{t_j}(\vec{u}, t) \wedge X_j(g_j(e_j, d), h_j(t, \vec{u}, \lambda))]\!]\eta'\delta[d \leftarrow e, \vec{u} \leftarrow \vec{v}, e_j \leftarrow e'_j, t \leftarrow t'] \\
&\quad \{\text{Definition of interpretation for existential quantification}\} \\
\iff &[\![\exists_{e_j : E_j}.\exists_{t:\mathrm{Real}}.(b \wedge \phi^{\wedge}_{t_j}(\vec{u}, t) \wedge X_j(g_j(e_j, d)), h_j(t, \vec{u}, \lambda)))]\!]\eta'\delta[d \leftarrow e, \vec{u} \leftarrow \vec{v}]
\end{aligned}
$$

By definition of disjunction this implies that Equation 6.3 holds.

Case $\phi_X \equiv \bigwedge_{i \in I} \phi^{\wedge}_{Z_i}$. By a derivation equivalent to the second case in Lemma 8 this does not depend on any predicate variables. As such, by the minimality of the proof graph, if this case was true the edge should not be present. So proven by contradiction. □

Note that equation (6.3) is the fulfilment property for a single successor. The next theorem establishes the existence of a set of paths in the proof graph of the original equation system whenever there is a path in the proof graph of the zoned equation system. For the purpose of proving this theorem the next definition defines a predecessor mapping *pre*.

Definition 45. Given a proof graph $\langle \tilde{V}, \tilde{\to} \rangle$ for a zoned equation system. Then, let $pre : 2^V \to 2^V$ be a mapping on signatures of the original proof graph that takes a set of vertices $S \subseteq V$ such that for every vertex $X(e, \vec{v}) \in pre(S)$ there is a vertex $Y(f, \vec{w}) \in S$ such that $X(e, \vec{v}) \to Y(e, \vec{w})$ satisfies the fulfilment property.

According to Lemma 9 it is always possible to find this mapping on vertices in the original proof graph whenever there is an edge in the zoned proof graph. Some additional path notation is used to prove the following theorem. Given a path $\pi = v_0 v_1 v_2 ...$, let π^i denote the sub-path starting with v_0 of length i. For a set of paths Π, let Π^i denote a set of sub-paths of $\pi \in \Pi$ of length i. Formally, the set Π^i is defined as $\{\pi^i \mid \pi \in \Pi\}$.

Theorem 6. *Let $\mathcal{E}$ be an equation system that satisfies the restrictions defined in Definition 43. Let $\tilde{G} = \langle \tilde{V}, \tilde{\to} \rangle$ be a minimal proof graph for equation system $T_Z(\mathcal{E})$. Then for every vertex $\tilde{X}(e, Z) \in \tilde{V}$ there exists an element $\vec{v} \in Z$ for which a proof graph $G = \langle V, \to \rangle$ can be constructed for $\mathcal{E}$ such that $X(e, \vec{v}) \in V$.*

Proof. Consider a path $\pi : \tilde{X}(e, Z) \to \tilde{X}_1(e_1, Z_1) \to ...$ in $\tilde{G}$. We show that there exist at least one path starting with $X(e, \vec{v})$ for some $\vec{v} \in Z$ that satisfies the fulfilment property for each of its edges.

This can be shown by induction on the length of all sub-paths of π. Let Π be a set of paths of vertices in V starting in vertices $X(e, \vec{v}_0)$ with $\vec{v}_0 \in Z_0$ such that for all edges in every path in

Π the fulfilment property holds. The induction hypothesis states that for i, there exist a set of paths Π^i such that every path in Π^i satisfies the fulfilment property for all of its transitions and for every $X_i(e_i, \vec{v}_i)$ with $\vec{v}_i \in Z_i$ there is a path in Π^i ending in that vertex.

Case i equals zero. There are no edges in Π^0 and so the statement holds vacuously.

The inductive step. Take an arbitrary path π^i in Π^i. Assume that π^i has a successor and as such we must show that Π^{i+1} exists. First, let Φ^{i+1} be a set of all possible vertices $X_{i+1}(e_{i+1}, \vec{v}_{i+1}) \in sig(\mathcal{E})$ such that $\vec{v}_{i+1} \in Z_{i+1}$. Let Φ^i denote the set of vertices in the last vertex of every path in Π^i. The projection $pre(\Phi^{i+1})$ must be a subset of Φ^i. The set of paths Π^{i+1} can be established by extending paths ending in $pre(\Phi^{i+1})$ to Φ^{i+1} and removing all paths ending with a vertex in the set $\Phi^i \setminus pre(\Phi^{i+1})$. The set Π^{i+1} cannot be empty, because $pre(\Phi^{i+1})$ is not empty. The fulfilment property is satisfied and every vertex in Φ^{i+1} is reachable because of the way that $pre(\Phi^{i+1})$ is defined.

In case that π is finite and has length n there is at least one sink $X_n(e_n, \vec{v}_n) \in \Phi^n$ by Lemma 8. The graph $G = \langle V, \rightarrow \rangle$ can be constructed by adding one path in Π^n that ends in $X_n(e_n, \vec{u}_n) \in \Phi^n$. Otherwise, graph G can be constructed by adding all vertices and edges for any path in $\lim_{i \to \infty} \Pi^i$. The resulting graph G satisfies the fulfilment property by the invariants of Π. The proof graph property is satisfied because the fixed point sequence and ranks of path π and any path in Π is the same according to Lemma 2. Therefore G is a valid proof graph for the signature $X(e, \vec{v})$ in equation system $\mathcal{E}$. □

6.2.2 Constructing the zoned proof graph

Lemma 10. *Let $\mathcal{E}$ be an equation system that satisfies the restrictions defined in Definition 43. Let $\tilde{\mathcal{E}}$ be its corresponding zoned equation system equal to $T_Z(\mathcal{E})$. Let $G = \langle V, \rightarrow \rangle$ be a minimal proof graph for $\mathcal{E}$. For every edge $X(e, \vec{v}) \rightarrow Y(f, \vec{w})$ it holds that for all signatures $\tilde{X}(e, Z) \in sig(T_Z(\mathcal{E}))$ where $\vec{v} \in Z$ holds there exists a signature $\tilde{Y}(f, Z') \in sig(\tilde{\mathcal{E}})$ with $\vec{w} \in Z'$ where Equation (6.7) holds.*

$$[\![\phi_{\tilde{X}}]\!]\eta[sig(T_Z(\mathcal{E})) \mapsto \mathit{false}][\{\tilde{Y}(f, Z')\} \mapsto \mathit{true}]\delta[d \leftarrow e, z \leftarrow Z] \tag{6.7}$$

Proof. Take an arbitrary edge $X(e, \vec{v}) \rightarrow Y(f, \vec{w})$ and an arbitrary signature $\tilde{X}(e, Z) \in sig(T_Z(\mathcal{E}))$ such that $\vec{v} \in Z$ holds. Proof by case distinction on the structure of ϕ_X.

Case $\phi_X \equiv \bigvee_{i \in I} \phi_X$. From the minimality of G and definition of disjunction it can be inferred that there exists an $j \in I$ such that the following interpretation yields true. Let η' be a valuation defined as $\eta[sig(\mathcal{E}) \mapsto \text{false}][\{Y(f, \vec{w})\} \mapsto \text{true}]$ and X_j is equal to Y.

$$\begin{aligned}
&[\![\exists_{e_j:E_j}.\exists_{t:\text{Real}}.(b \wedge \phi^{\wedge}_{t_j}(\vec{u}, t) \wedge Y(g_j(e_j, d), h_j(t, \vec{u}, \lambda))]\!]\eta'\delta[d \leftarrow e, \vec{u} \leftarrow \vec{v}] \\
&\quad \{\text{Definition of interpretation for existential quantification}\} \\
\iff &\exists_{e'_j \in \mathbb{E}_j}\exists_{t' \in \mathbb{R}}[\![b \wedge \phi^{\wedge}_{t_j}(\vec{u}, t) \wedge Y(g_j(e_j, d), h_j(t, \vec{u}, \lambda))]\!]\eta'\delta[d \leftarrow e, \vec{u} \leftarrow \vec{v}, e_j \leftarrow e'_j, t \leftarrow t']
\end{aligned}$$

Existential elimination, pick values $e'_k \in \mathbb{E}_j$ and $q \in \mathbb{R}$ such that this interpretation holds. As such $g_j(e'_k, e)$ equals f and $h_j(\vec{v}, q, \lambda)$, for some $\lambda \subseteq \mathbb{N}$, equals $\vec{w}$. Let $\tilde{\eta}$ be a valuation defined as $\eta[sig(\tilde{\mathcal{E}}) \mapsto \text{false}][\{(\tilde{Y}, f, Z'\} \mapsto \text{true}]$. Derivation for boolean term b:

$$\begin{aligned}
&[\![b]\!]\eta'\delta[d \leftarrow e, \vec{u} \leftarrow \vec{v}, e_j \leftarrow e'_k, t \leftarrow q] \\
&\quad \{\text{Definition of } b\text{, does not depend on } t \text{ and } \vec{u}\} \\
\iff &[\![b]\!]\eta'\delta[d \leftarrow e, e_j \leftarrow e'_j]\eta'\delta[d \leftarrow e, e_j \leftarrow e'_k] \\
&\quad \{\text{Definition of } b\text{, does not depend on } z \text{ and valuations}\} \\
\iff &[\![b]\!]\eta'\delta[d \leftarrow e, e_j \leftarrow e'_j]\tilde{\eta}\delta[d \leftarrow e, z \leftarrow Z, e_j \leftarrow e'_k]
\end{aligned} \tag{6.8}$$

Then the interpretation of $\phi^{\wedge}_{t_j}(\vec{u}, t)$ yields true for given $\vec{v}$ and q so by Lemma 7 the following

derivation holds:

$$
\begin{aligned}
& [\![\phi^{\wedge}_{t_j}(\vec{u}, t)]\!]\eta'\delta[d \leftarrow e, \vec{u} \leftarrow \vec{v}, e_j \leftarrow e'_k, t \leftarrow q] \\
& \{\text{Lemma 7 and definition } \texttt{is_notempty}\text{, does not depend on valuation}\} \\
\iff & [\![\texttt{is_notempty}(T_z(z, \phi^{\wedge}_{t_j}))]\!]\tilde{\eta}\delta[d \leftarrow e, z \leftarrow Z, e_j \leftarrow e'_k] \qquad (6.9)
\end{aligned}
$$

By definition $g_j(e'_k, e)$ equals f in both cases. Let Z' be the result of $\texttt{reset}(T_z(z, \phi^{\wedge}_{t_j}), \lambda))$. The remaining proof obligation is to show that $\vec{w} \in Z'$. By Lemma 6 the vector $\vec{v} + q$ resulting from the time-elapse satisfies the constraints of $\phi^{\wedge}_{t_j}$. By definition of the function `reset` the operation $(\vec{v} + q)[\lambda \leftarrow 0]$ is applied, which definition exactly matches the updates performed by h_j applied to $\vec{v}$, q and λ which is equal to $\vec{w}$, so $\vec{w} \in Z'$. Note that this derivation only holds one way, as such this lemma could not be used both ways. It does not hold that for any $\vec{v} \in Z$ it is possible to pick a t such that $\vec{v} + t$ occurs in the result of $T_z(z, \phi^{\wedge}_{t_j})$.

Then the following interpretation is true by definition:

$$
\begin{aligned}
& \{\text{By } f \text{ and } Z' \text{ observations this holds by definition}\} \\
& \tilde{\eta}(\tilde{Y})(\delta[d \leftarrow e, z \leftarrow Z, e_j \leftarrow e'_k](g_j(e_j, d)), \delta[d \leftarrow e, z \leftarrow Z, e_j \leftarrow e'_k](\texttt{reset}(T_z(z, \phi^{\wedge}_{t_j}), \lambda)))) \\
& \{\text{Definition of interpretation for predicate variables}\} \\
\iff & [\![\tilde{Y}(g_j(e_j, d), \texttt{reset}(T_z(z, \phi^{\wedge}_{t_j}), \lambda))]\!]\eta'\delta[d \leftarrow e, z \leftarrow Z, e_j \leftarrow e'_k] \\
& \{\text{Does not depend on } t\} \\
\iff & [\![\tilde{Y}(g_j(e_j, d), \texttt{reset}(T_z(z, \phi^{\wedge}_{t_j}), \lambda))]\!]\eta'\delta[d \leftarrow e, z \leftarrow Z, e_j \leftarrow e'_k,] \qquad (6.10)
\end{aligned}
$$

Finally, these observations can be combined. Note that $\tilde{Y}$ equals $\tilde{X}_j$:

$$
\begin{aligned}
& \{\text{Derivations 6.8, 6.9 and 6.10, definition of conjunction and existential introduction}\} \\
\implies & \exists_{e_j \in \mathbb{E}_j} [\![b \wedge \texttt{is_notempty}(T_z(z, \phi^{\wedge}_{t_j})) \wedge \tilde{X}_j(g_j(e_j, d), \texttt{reset}(T_z(z, \phi^{\wedge}_{t_j}), \lambda))]\!] \\
& \quad \tilde{\eta}\delta[d \leftarrow e, z \leftarrow Z, e_j \leftarrow e'_j] \\
& \{\text{Definition of interpretation for existential quantification}\} \\
\iff & [\![\exists_{e_j : E_j}.(b \wedge \texttt{is_notempty}(T_z(z, \phi^{\wedge}_{t_j})) \wedge \tilde{X}_j(g_j(e_j, d), \texttt{reset}(T_z(z, \phi^{\wedge}_{t_j}), \lambda)))]\!]\tilde{\eta}\delta[d \leftarrow e, z \leftarrow Z]
\end{aligned}
$$

By definition of disjunction this implies that Equation (6.7) holds.

Case $\phi_X \equiv \bigwedge_{i \in I} \phi^{\wedge}_{Z_i}$. By a derivation equivalent to the second case in Lemma 8 this does not depend on any predicate variables. As such, by the minimality of the proof graph, if this case was true the edge should not be present. So proven by contradiction.

□

Theorem 7. *Let $\mathcal{E}$ be an equation system that satisfies the restrictions defined in Definition 43. Let $G = \langle V, \rightarrow \rangle$ be a minimal proof graph for $\mathcal{E}$. Then for every vertex $X(e, \vec{v}) \in V$ there exists a proof graph $\tilde{G} = \langle \tilde{V}, \tilde{\rightarrow} \rangle$ for equation system $T_Z(\mathcal{E})$ such that $\tilde{X}(e, Z) \in \tilde{V}$ and $\vec{v} \in Z$.*

Proof. Consider a path $\pi : X(e, \vec{v}) \rightarrow X_1(e_1, \vec{v}_1) \rightarrow ...$ in G. We show that it is possible to define a path $\tilde{\pi}$ between signatures in $T_Z(\mathcal{E})$ such that every edge in that path satisfies the fulfilment property. This is shown by natural induction on the length of a sub-path π^i of π. The induction hypothesis states that for a given π^i there exists a path $\tilde{\pi}^i$ such that each edge satisfies the fulfilment property and for every vertex $\tilde{X}_i(e_i, Z_i)$ it holds that $\vec{v}_i \in Z_i$.

Case i equals zero. Let $\tilde{\pi}$ be a path consisting of only π^0. There are no edges in $\tilde{\pi}^0$ so the fulfilment property holds vacuously and $\vec{v} \in Z$ by definition.

Inductive step, take a path π^i. If there is a path π^{i+1}, by Lemma 10 because $\vec{v}_i \in Z_i$ by the induction hypothesis there is a successor signature $(\tilde{X}_{i+1}, e_{i+1}, Z_{i+1})$ such that an edge from π_i to that vertex satisfies the fulfilment property. This implies that path $\tilde{\pi}^i$ can be extended with

signature $\tilde{X}_{i+1}(e_{i+1}, Z_{i+1})$ obtaining path $\tilde{\pi}^{i+1}$ such that for every edge in this new path the fulfilment property holds and $\vec{v}_{i+1} \in Z_{i+1}$ by Lemma 10.

Now by adding all the vertices and transitions of path $\tilde{\pi}$ to $\tilde{G}$ a proof graph is obtained. In case that π was finite Lemma 8 implies that the last vertex in $\tilde{\pi}$ does not require a successor, because the last vertex in π did not require a successor as well. The proof graph property is satisfied, because the sequence of fixed points and ranks of vertices in π and $\tilde{\pi}$ are the same according to lemma 2. As such $\tilde{G}$ is a valid proof graph for signature $X(e, Z)$ in the zoned equation system. □

6.2.3 Combination

Finally. by combining the results of proof graph constructions in both ways it can be established that Theorem 5 holds.

Theorem 8. *Let $\mathcal{E}$ be an equation systems that satisfies the restrictions of Definition 43. For all predicate variables $X \in bnd(\mathcal{E})$ there is a zone Z such that the following equation holds:*

$$\exists_{\vec{v} \in Z}([\![\mathcal{E}]\!](X)(e, \vec{v})) \iff [\![T_Z(\mathcal{E})]\!](\tilde{X})(e, Z) \tag{6.11}$$

Proof. Case, assume that $[\![T_Z(\mathcal{E})]\!](\tilde{X})(e, Z)$ holds. By Theorem 6 it is possible to construct a proof graph $G = \langle V, \rightarrow\rangle$ such that $X(e, \vec{v}) \in V$ for some $\vec{v} \in Z$. Then by Definition 18 the solution of the interpretation $[\![\mathcal{E}]\!](X)(e, \vec{v})$ yields true.

Case, assume that $[\![\mathcal{E}]\!](X)(e, \vec{v})$ holds for some $\vec{v} \in Z$. By Theorem 7 it is possible to construct a proof graph $\tilde{G} = \langle \tilde{V}, \rightarrow\rangle$ such that $\tilde{X}(e, \vec{v}) \in \tilde{V}$ and $\vec{v} \in Z$. Then by Definition 18 the solution of the interpretation $[\![T_Z(\mathcal{E})]\!](\tilde{X})(e, Z)$ yields true. □

This concludes the proof of correctness which shows that the presented zoned equation system yields a suitable, but weaker, abstraction for a subset of equation systems with real-valued parameters. Similar to the result for regions this theorem shows that the solution for the signature $X(1, 0, 0)$ in the equation system of Example 4 is true, because the solution for signature $X(1, \{(0, 0)\})$ in the zoned equation system of Example 10 yields true.

6.3 Lasso proof graph

In Theorem 6 there is another case where the zone abstraction proof graph contains a lasso. A lasso is a path that consists of a unique prefix followed by an infinitely repeated suffix. This means that the resulting proof graph is finite, as for disjunctive formulas only a single path is important. Here, we show a counter example that shows that this does not necessarily mean that the original proof graph is also finite, and as such this case had to be considered as an infinite path. Given an equation system:

$$\nu X(x, y : \text{Real}) = \exists_{t:\text{Real}}.(y + t > 0 \land X(x + t, 0))$$

Now abstract it using the zone translation rules, obtaining:

$$\begin{aligned}\nu \tilde{X}(z : \text{Zone}) = \ & \texttt{is_notempty}(\texttt{conjunction}(\texttt{time_elapse}(z), y + t > 0)) \\ & \land \tilde{X}(\texttt{reset}(\texttt{conjunction}(\texttt{time_elapse}(z), y + t > 0)), \{1\}))\end{aligned}$$

The instantiated boolean equation system together with the proof graph shown on the right:

$$\begin{aligned}\nu \tilde{X}_Z &= \tilde{X}_{Z_1} \\ Z &= \{0\} \\ \nu X_{Z_1} &= X_{Z_1} \\ Z_1 &= \{(x + t, 0) \mid x \in Z, t \in \mathbb{R}, y + t > 0\}\end{aligned}$$

With the resulting proof graph being finite:

$$X_Z \rightarrow X_{Z_1} \circlearrowleft$$

Figure 6.1: The corresponding proof graph

In the original however, starting with $x = 0$ as a starting point it becomes an infinite chain of increasing real-valued values.

$$\begin{aligned} \nu\tilde{X}_0 &= \tilde{X}_1 \\ \nu X_1 &= X_{9/4} \\ \nu X_{9/4} &= X_\pi \\ &\dots \end{aligned}$$

With the resulting proof graph being infinite:

$$X_0 \rightarrow X_1 \rightarrow X_{9/4} \rightarrow X_\pi \longrightarrow$$

Figure 6.2: The corresponding infinite proof graph

This implies that a lasso must also be considered as an infinite path.

Chapter 7

Implementation

7.1 mCRL2

The mCRL2 toolset implements a very rich language called minimal Common Representation Language to specify processes with data. For defining the data in equation systems only the data specification part of this language is important. For defining equation systems a very simple syntax is introduced that is very similar to the one defined in this thesis, but it uses ASCII symbols as opposed to mathematical ones for practical purposes. In this section we provide an introduction for the data specification part of the language and specifically for the subset that is relevant for the implementation. For the full language definition, see [9].

A data specification consists of a number of non-empty sorts, a number of *constructors* for each sort, a number of *mappings* and a number of equations. Constructors and mappings are both functions, with the distinction that constructors define the elements in the sort, but mappings are only allowed to *alias* existing elements. This means that mappings cannot define new elements, but only use constructors for that purpose. A mapping is defined by a name, a number of input sorts separated by # and a single output sort. Each mapping is defined by a number of equations that specify the rewrite rules for that mapping, associated by having the same name and sorts.

Consider the following example mapping `fib: Nat -> Nat;` which maps natural numbers to natural numbers. This mapping specifies the Fibonacci sequence. Its implementation is defined by two equations:

```
n <= 1 -> fib(n) = n;
n > 1  -> fib(n) = fib(Int2Nat(n - 1)) + fib(Int2Nat(n - 2))
```

Each equation defines a rule that the term rewrite system can use to rewrite, for example the term `fib(3)`. The part of each equation before the right arrow defines an *enabling* condition. This means that only the second equation can be used to rewrite `fib(3)`, as three is not less or equal to one. As such applying the second equation to `fib(3)` yields `fib(2) + fib(1)`. These can be rewritten again to `fib(1) + fib(0) + fib(1)`, which can be rewritten to `1 + 0 + 1`. Then the addition is applied yielding the term `2`. Note that the mapping `Int2Nat` is used to change the sort of the result of $n - 1$ from an integer to a natural number. This is required because the result of a subtraction yields an integer, because for example $3 - 5$ yields -2. Finally for the specification note that all sorted variables are defined in the specification, which in this case is `n : Nat`. However, this will be omitted from our implementation details, because the sort can be inferred from the parameters.

From the pre-defined sorts we will use the sorts representing natural numbers $\mathbb{N}$ and booleans $\mathbb{B}$, denoted by Nat and Bool respectively. For these sorts many standard arithmetic and logical functions are defined. We will also use lists over a sort D, which are denoted by List(D). A list is defined by the constructor [], which is the empty list, and given an element a of sort D and a list b of sort List(D) then the constructor $a \triangleright b$ which prefixes element a to list b. The mapping

`head` is defined for a list that returns the element on the first index of the list and the mapping `tail` is defined for a list that returns a list with its head removed. Finally, a mapping denoted by `.` is defined to return the element at a specific index. This mapping is written in infix notation, i.e. for a given list b constructed as `[1,6,4,4,3]` the application `b . 4` yields `3`.

For our own sorts we will only be using so-called *structured* sorts. A structured sort is denoted by `struct A = a(d : D) | b(e : E) | ...`. This construction results in a number of automatically generated constructors named $a(d : D)$, $b(e : E)$... that allow us to construct elements of sort A with specific values, for example $a(d')$, where d' is an element of sort D. The value used for construction can be obtained by applying $d(a)$ to obtain d'. Furthermore the structured sort defines a set of recogniser functions that return true when an element was constructed with a given constructor, for example `is_a(a')` would yield true if and only if a' was constructed using the a constructor.

Note that sorts can only be constructed using the constructors and there are no constructors to modify a sort, as such the elements of every sort are immutable. As such the list can only be changed by creating a new list. The final language construct that we will use is only a short-hand which is represented by the `whr ... end` clause. This notation allows us to define a local variable that can be used within the equation definition.

7.2 Regions

Regions can be represented in a rather straightforward way. Each region is represented by one list containing one integral constraint for each real-valued variable in $\vec{u}$ and a second list representing the fractional ordering of these variables.

Remark 7. In our implementation each variable is identified by its position in $\vec{u}$. For example, given the parameters $u_0, \cdots, u_k$ then variable u_0 is identified by the natural number zero. The reason for this is that it is easier to store indices as opposed to the "actual" variable name. In the explanation we refer to the variable name directly and leave out this detail from the description.

First, the fractional ordering representation is defined. This representation must support the ability to (quickly) obtain the set of variables with the least fractional value and be able to remove the fractional ordering for variables with an equal to or unbounded constraint. The following representation makes it easy to implement this functionality. Essentially, a list of sets is used where the position in the list defines the order between the sets; and all variables in the same set have the same fractional value. Formally defined as:

Definition 46. Given a list of real-valued variables $\vec{u}$. Let $Lf = [S_0, \cdots, S_k]$ be a list such that each S_i is a set of indices, i.e., for all $i \in \mathbb{N}$ and $0 \leq i < k$ it holds that $S_i \subseteq \mathbb{N}$. For a list Lf to represent a fractional ordering the following conditions must hold:

- Each natural number is the index of a variable in $\vec{u}$. Formally, for all $0 \leq i < k$ it holds that for all $x \in S_i$ implies $0 \leq x < |\vec{u}|$.
- Every index can only belong to one set in this list. Formally, for all $j \in \mathbb{N}$ and $0 \leq i, j < k$ and $i \neq j$ it holds that $S_i \cap S_j$ equals $\emptyset$.

The meaning of the indices in this list is defined as follows:

- For any two variables $x, y \in S_i$ it holds that their fractional values are the same, i.e. $fr(x)$ equals $fr(y)$.
- The position of a set determines the fractional ordering between the sets. Formally, for all $i, j \in \mathbb{N}$ and $0 \leq i < j < k$ it holds that if $x \in S_i$ and $y \in S_j$ then $fr(x)$ is less than $fr(y)$.

Note that in this definition the set of variables with the least ordering is given by S_0 and removing the fractional ordering can be done by removing a variable from every set in the list.

Next, an example of how this representation can be used to describe the fractional ordering of a region.

Example 11. Consider the following real-valued variables x, y, z with values 1.7, 2.1 and 3.7 respectively. Their fractional ordering is given by $fr(y) < fr(x)$, $fr(x) < fr(z)$ and $fr(x) = fr(z)$, which can be represented by the list $[\{y\}, \{x, z\}]$.

First we define the sort that represents integral constraints. The integral constraint for each variable is represented by the structured sort `RegionBound` that has three constructors `equal(c : Nat)`, `between(c : Nat)` and `unbounded`. Now a region can be defined as the combination of two list as presented in the next definition. The regions itself are represented by a structured sort called `Region` with the constructor `region` that has `bounds : List(RegionBound)` and `order : List(List(Nat))` as parameters for the integral constraints and the fractional ordering.

Note that for the fractional ordering we use a list to represent a finite set as opposed to using the built-in set sort. The built-in set sort represents infinite sets by a characteristic function. There is no way to pick an arbitrary element from this set representation, the only mapping provided is to check whether a specific element is in a set. This means that there is no convenient way to iterate over the elements in such a set, which made it unsuitable for our purpose.

The functions that are defined for the regions are now defined as a mapping and a set of equations specified as an mCRL2 data specification. In the implementation of the functions we use a number of auxiliary mappings to improve readability. The following mapping takes a region and an index representing a variable and returns the integral bound for that variable.

```
get_bound : Region # Nat -> RegionBound;
```

It is defined by the single equation `get_bound(reg, i) = bounds(reg) . i`. For updates to the elements in a list the following mapping is introduced which constructs a new list with a single integral bound changed

```
set : List(RegionBound) # Nat # RegionBound -> List(RegionBound);
```

The natural number indicates the position in the list that should be updated. Each iteration the head of the region bound list is taken and added to the output, the position that should be updated is decreased. The first equation defines the case where the current position is zero, in which case the updated bound is added to the front of the remaining original list. The second equation defines the iteration case.

```
           set(regbnd |> regbnds, 0, newbnd) = newbnd |> regbnds;
(i > 0) -> set(regbnd |> regbnds, i, newbnd) = regbnd
            |> set(regbnds, prev(i), newbnd);
```

Here we see a typical construct used to iterate over the (some) elements of a list. The construct `regbnd |> regbnds` is used to pattern match a list of size greater than zero. Its head is matched to `regbnd` and its tail to `regbnds`. The first equation defines the case where the index i is zero in which case the current position is replaced by `newbnd` and the tail of the list is kept the same. The second equation defines the case where the current index is not the index that we want to update. In that case the index is decremented by the mapping `prev(i) : Nat -> Nat` which is defined by the equation `(i > 0) -> prev(i) = Int2Nat(i - 1)` or the predecessor of a natural number. Note that the term `Pred(0)` cannot be rewritten and as such results in an error. Similarly a mapping `succ(i)` is defined for the successor. In this case the head `regbnd` is kept and the `set` mapping is applied to the tail of the matched list.

Note that the fact that terms are immutable means that updating a list takes $O(n)$ worst case, where n is the number of elements in the list. The `set` mapping mapping can be extended to a mapping updating integral constraints in a region, this is defined as:

```
set : Region # Nat # RegionBound -> Region
```

Which is defined by the equation:

```
set(reg, i, regbnd) = region(set(bounds(reg), i, regbnd), order(reg));
```

Now the operations on regions that are defined for the translation are specified as a number of mappings and a number of equations in the mCRL2 language.

Satisfies

The satisfies function is implemented by the mapping:

```
satisfies : Region # Nat # Expression -> Bool
```

The `Expression` sort is defined as the structured sort `Expression` with constructors `less(c : Nat)`, `lessEqual(c : Nat)`, `greater(c : Nat)` and `greaterEqual(c : Nat)` which denote all possible comparisons of a natural number and a real-valued variable. The mapping is specified by a number of equations `satisfies(reg, x, expression)`, where `reg` is a region, `x` is a natural number specifying the real-valued parameter and `expression` is of sort `Expression`, that are enabled depending on the constructor of the integral bound specified for `x` and the constructor of the expression used. Now the function can be implemented by considering a number of cases.

For unbounded integral constraints the expression less or less equal always result in false and greater and greater equal results in true. Defined by the following equations:

```
is_unbounded(get_bound(reg, x)) && (is_less_equal(expression)
    || is_less(expression)) -> satisfies(reg, x, expression) = false;
is_unbounded(get_bound(reg, x)) && (is_greater_equal(expression)
    || is_greater(expression)) -> satisfies(reg, x, expression) = true;
```

For equal integral constraints the constant of the constraint is simply compared with the constant used the expression and the type of comparison depends on the constructor used in the expression. This case is defined by the following equations:

```
is_equal(get_bound(reg, x)) && is_less_equal(expression)
    -> satisfies(reg, x, expression) = c(get_bound(reg, x)) <= c(expression);
is_equal(get_bound(reg, x)) && is_less(expression)
    -> satisfies(reg, x, expression) = c(get_bound(reg, x)) < c(expression);
is_equal(get_bound(reg, x)) && is_greater(expression)
    -> satisfies(reg, x, expression) = c(get_bound(reg, x)) > c(expression);
is_equal(get_bound(reg, x)) && is_greater_equal(expression)
    -> satisfies(reg, x, expression) = c(get_bound(reg, x)) >= c(expression);
```

For between integral constraints the highest or lowest bounds are compared with the constraint in the expression and the type of comparison depends on the constructor used in the expression:

```
is_between(get_bound(reg, x)) && is_less_equal(expression)
    -> satisfies(reg, x, expression) = c(get_bound(reg, x)) <= c(expression);
is_between(get_bound(reg, x)) && is_less(expression)
    -> satisfies(reg, x, expression) = c(get_bound(reg, x)) <= c(expression);
is_between(get_bound(reg, x)) && is_greater(expression)
 -> satisfies(reg, x, expression) = (c(get_bound(reg, x)) - 1) >= c(expression);
is_between(get_bound(reg, x)) && is_greater_equal(expression)
 -> satisfies(reg, x, expression) = (c(get_bound(reg, x)) - 1) >= c(expression);
```

This concludes the description of the satisfies function, which shows that each timing constraint can be checked in constant time and only depends on the integral constraint.

Reset

For the purpose of implementing the `reset` function two mappings are introduced. To set a region a number of variables are set to an `equal(0)` bound and their fractional ordering is removed. The following mapping takes the fractional ordering and a list of indices and removes every variable that should be reset from every list in the fractional ordering, it is defined as:

```
remove_fractions : List(List(Nat)) # List(Nat) -> List(List(Nat));
```

Its implementation is defined as an iteration over every list in the fractional ordering. Where for every list the variables that should be reset are removed. When the resulting list is empty it is also removed from the list of lists, this is done to make sure that the fraction list does not contain any empty lists. Its implementation is defined by the following equations:

```
                                   remove_fractions([], selection) = [];
setminus(frac, selection) != [] -> remove_fractions(frac |> fracs, selection)
    = setminus(frac, selection) |> remove_fractions(fracs, selection);
setminus(frac, selection) == [] -> remove_fractions(frac |> fracs, selection)
    = remove_fractions(fracs, selection);
```

This is another example of an iteration. The first equation defines the case where the fractional ordering is the empty list. In the two other equations the variables in the selection are removed by the call to `setminus` which takes two lists as input and returns a list where every elements in the second list is removed from the first list. When the result of `setminus` is empty, which is in the enabling condition of the third equation, it is also removed from the fraction list. Otherwise the potentially reduced list is kept in the fraction list.

The next mapping sets a number of integral constraints to `equal(0)`, it is defined as:

```
reset_bounds : List(RegionBound) # List(Nat) -> List(RegionBound);
```

Its implementation loops over every variable that should be reset and changes their integral bound. It is implemented by the equations:

```
reset_bounds(regbnds, []) = regbnds;
reset_bounds(regbnds, i |> selection)
    = reset_bounds(set(regbnds, i, equal(0)), selection);
```

The first equation is the base case, where the `selection` variable is the empty list. The second equation is enabled while the selection is not empty. Each iteration the head of the list matched is used to reset an integral bound to be equal to zero and the tail is used in the recursive call such that the head becomes the next element in the list.

Finally, the reset function is implemented by the following mapping:

```
reset : Region # List(Nat) -> Region;
```

This mapping can be implemented by a single call to the previously defined functions, resulting in:

```
reset(reg, selection) = region(reset_bounds(bounds(reg), selection),
    remove_fractions(order(reg), selection));
```

Time successors

For the purpose of implementing the function `is_successor` two mappings are introduced. One mapping is introduced that returns the set of regions for a given region and another mapping for checking whether a region is an element of this set. The first mapping requires several functions that compute the set of variables which have specific integral bounds. These mappings are given by:

```
get_variables_equal : List(RegionBound) -> List(Nat);
get_variables_between : List(RegionBound) -> List(Nat);
```

These mappings return a list of indices of variables with an `equal` bound, or `between` bound respectively. Each mapping is implemented as a loop over all bounds that decides whether the current index has the desired bound using the associated recognizer function. The `get_variables_equal` mapping is implemented by the equations:

```
                    get_variables_equal([], i) = [];
 is_equal(regbnd) -> get_variables_equal(regbnd |> regbnds, i)
    = i |> get_variables_equal(regbnds, succ(i));
!is_equal(regbnd) -> get_variables_equal(regbnd |> regbnds, i)
    = get_variables_equal(regbnds, succ(i));
```

These equations specify an iteration over all integral bounds, denoted by `regbnds`, and check whether the current bound is constructed using the `equal` constructor. Each iteration when this is the case the current index is added to the returned list.

The equations for `get_variables_between` are the same, except that they check for the `is_between` constructor. Finally, another mapping is introduced that returns true whenever all integral constraints are unbounded:

```
is_unbounded : List(RegionBound) -> Bool
```

This is achieved by an iteration over all bounds which returns false whenever one is not unbounded. Which is implemented by the equations:

```
                        is_unbounded([]) = true;
is_unbounded(regbnd) -> is_unbounded(regbnd |> regbnds) = is_unbounded(regbnds);
!is_unbounded(regbnd) -> is_unbounded(regbnd |> regbnds) = false;
```

Each of these mappings is then extended to a mapping over regions. This is achieved by introducing a mapping with the same name that takes a region and returns the application of the relevant mapping with `bounds(reg)` as parameter.

First, the mapping `time_successor : Region # List(Nat) -> Region` is defined that computes the closest time-successor for the current region. This is implemented as a number of cases.

If for every real-valued variable $x \in \vec{u}$ it holds that the constraint in R satisfies $(x > c_x)$, then the only time successor $\Uparrow R$ equals to R. In this region the time can progress for ever without changing the result for any b_c that it satisfies or does not. This is defined by the following equation:

```
(is_unbounded(reg)) -> time_successor(reg, upperbnds) = reg;
```

Now suppose that the set C_0 consisting of variables $x \in C_0$ such that the region R satisfies the constraint $(x = c)$ for some $c \leq c_x$, is non-empty. These are the indices that result from `get_variables_equal`. Then the time successor $\Uparrow R$ of R is defined as:

- For variables $x \in C_0$ if region R satisfies $(x = c_x)$ then the time successor satisfies $(x > c_x)$.
- For variables $x \in C_0$ if region R satisfies $(x = c)$ then the time successor satisfies $(c - 1 < x < c)$ and for $x \notin C_0$ the constraints for the time successor are the same.
- For all clocks x and y such that R satisfies $x \leq c_x$ and $y \leq c_y$, the fractional ordering in the time-successor between $fr(x)$ and $fr(y)$ stays the same.

For this case a mapping is introduced to set the fraction ordering of a number of variables to be the smallest possible. This is required for the case that $(x = c)$ and the successor is a between constraint. The mapping is specified as:

```
reset_fractions : List(List(Nat)) # List(Nat) -> List(List(Nat));
```

Its implementation removes the fraction from all other lists using the previously defined mapping `remove_fractions` and prepends the list of new smallest fractions, which are passed as argument, to the front. It is defined by the following equation:

```
reset_fractions(fracs, selection) = selection
    |> remove_fractions(fracs, selection);
```

Now the case for the time-successor can be implemented by the following mapping.

```
time_successor_equal : Region # List(Nat) # List(Nat) -> Region
```

Its arguments are the regions, a list of upper bounds and a selection of variables with the equal to constraint. The list of upper bounds $c_{\vec{u}_i}$ must match the length of the integral constraint list. In other words for every variable an upper bound must be defined. The mapping is defined by the following equations:

```
time_successor_equal(reg, upperbnds, []) = reg;
(c(get_bound(reg, i)) != upperbnds . i)
    -> time_successor_equal(reg, upperbnds, i |> selection)
        = time_successor_equal(reset_fractions(
            set(reg, i, between( succ(c(get_bound(reg, i))) )), [i]),
          upperbnds, selection);

(c(get_bound(reg, i)) == upperbnds . i)
    -> time_successor_equal(reg, upperbnds, i |> selection)
        = time_successor_equal(remove_fractions(
            set(reg, i, unbounded), [i]),
          upperbnds, selection);
```

The first equation defines the case where the selection is empty. The second equation defines the case where the head variable in the selection has a constraint $(x = c)$ and the last equation defines the case where the head variable in the selection has a constraint $(x = c_x)$.

A second equation for `time_successor` calls this mapping in the case that C_0 is not empty, which is equal to `get_variables_equal(reg) != []`.

```
(get_variables_equal(reg) != []) -> time_successor(reg, upperbnds)
    = time_successor_equal(reg, upperbnds, get_variables_equal(reg));
```

If the both cases do not apply. Let C_0 be a set of variables for which region R does not satisfy $(x > c_x)$ and have maximal fractional parts. This means that for every variable y which does not satisfy $(y > c_y)$ it holds that $fr(y) \leq fr(x)$. In this case when time progresses these variables will assume integer values. The time-successors are the region R itself and a region defined as:

- For $x \in C_0$ if R satisfies $(c-1 < x < c)$ then $\Uparrow R$ satisfies $(x = c)$. For $x \notin C_0$ the constraints are the same.
- For all clocks x and y such that R satisfies $(c-1 < x < c)$ and $(d-1 < x < d)$, the fractional ordering between $fr(x)$ and $fr(y)$ stays the same.

This case is implemented by the mapping:

```
time_successor_between : Region # List(Nat) -> Region
```

This mapping is defined by the following equations:

```
time_successor_between(reg, []) = reg;
time_successor_between(reg, i |> selection) = time_successor_between(
    set(reg, i, equal( c(get_bound(reg, i))) ), selection);
```

This represents a loop where for each iteration an variable from selection is changed from a between to an equal constraint. To obtain the set C_0 the indices of variables must be known with maximal fractional ordering. For this purpose the following mapping is introduced:

```
get_maximal_fractions : List(List(Nat)) -> List(Nat);
```

Its implementation simply takes the last list in the fractional ordering, which by definition contains the indices of variables with the highest fractional ordering. This is implemented by the following equation:

```
(fracs != []) -> get_maximal_fractions(fracs) = rhead(fracs);
```

The mapping `rhead` is the reverse head, which returns the last element in a list.

The set C_0 is then computed by taking the intersection between the set of maximal fractions and the set of variable with an `between` constraint. The last case of the time-successor function can then be defined by the final equation:

```
(get_variables_between(reg) != [] && get_variables_equal(reg) == [])
    -> time_successor(reg, upperbnds)
        = time_successor_between(remove_fractions(reg, selection2), selection2)
            whr selection2 = intersection(get_maximal_fractions(reg),
                get_variables_between(reg)) end;
```

Now to compute the actual set of regions the mapping `time_successor` is called until it returns a region where every variable is unbounded, because this region only has itself as successor. Each region that it encounters is added to the return list, and we have to add the initial region to the set of regions. Now the function computing the set of regions, without the initial region, is defined as the mapping:

```
time_successors_continued : Region # List(Nat) -> List(Region);
```

This mapping adds the result of the `time_successor` mapping to a list and calls itself whenever the current region does not have an unbounded integral constraint for every variable. It is defined by the following equations:

```
(!is_unbounded(reg)) -> time_successors_continued(reg, upperbnds)
    = newreg |> time_successors_continued(newreg, upperbnds)
        whr newreg = time_successor(reg, upperbnds) end;
(is_unbounded(reg))  -> time_successors_continued(reg, upperbnds) = [];
```

Finally, the time-successors function can be defined by the mapping:

```
time_successors : Region # List(Nat) -> List(Region);
```

This adds the current region to the set of regions and calls the previously defined `time_successors_continued` mapping whenever the initial region was not unbounded. It is defined by the equations:

```
    (!is_unbounded(reg)) -> time_successors(reg, upperbnds)
        = reg |> time_successors_continued(reg, upperbnds);
    (is_unbounded(reg))  -> time_successors(reg, upperbnds) = [ reg ];
```

Finally, an element of function is given by the mapping defined as:

```
is_contained : Region # List(Region) -> Bool;
```

It is defined by the equation that simple calls the built-in function in, which returns true if and only if that element is part of the list.

```
is_contained(reg, regions) = reg in regions;
```

Then the `is_successor`(r', r) function can be used in the mCRL2 specification by the combination `is_contained(r', time_successors(r, upperbounds))`.

Initialize

Finally, a function that was not present in the translation, which initializes a region where every variable has an integral constraint equal to zero and no fractional ordering. This can be used to initialize regions $[x = y = 0]$ for a number of variables. It is defined as the mapping `initialize : Nat -> Region`. This mapping takes a natural number for the length of vector $\vec{u}$. It is defined by the equation:

```
initialize(n) = region(initialize_ints_for_i(n), []);
```

The mapping `initialize_ints_for_i : Nat -> List(RegionBound)` simply takes the length of the vector and creates a list of length n where each element is the integral constraint `equal(0)`. It is defined by the equations:

```
           initialize_ints_for_i(0) = [];
(i > 0) -> initialize_ints_for_i(i) = initialize_ints_for_i(prev(i))
                <| equal(0);
```

Which represent a loop from n to zero where each iteration `equal(0)` is appended.

7.3 Zones

For zones a slightly more complicated sort is used to represent them. Consider the following example zone over three variables x, y, z.

Example 12. For a vector x, y, z consider the following zone constraints:

$$x - y < 2 \wedge x - y \leq 5 \wedge y - z < 3$$

Notice that in the given zone constraint the condition $x - y \leq 5$ is unnecessary as $x - y < 2$ is a stronger bound. This observation leads to the notion of redundant constraints, which means that the required number of constraints to define a zone can be bounded. Two constraints $x - y \prec c$ and $x - y \prec d$, can be replaced by a single constraint comparing to the minimum of c and d. This means that for a vector of $\vec{u} : u_1 ... u_k$ variables there are at most k^2 difference constraints and as such these zones can be represented as a k by k matrix. These matrices are conveniently named difference bound matrices.

7.3.1 Difference bound matrices

It has been show before that similar clock constraints could be efficiently represented as difference bound matrices [3]. This representation is also used in the real-systems model checking tool UPPAAL [4]. For a matrix M let M_{ij} denote the element in row i and column j, starting with index zero.

Definition 47. Given a zone over k real-valued variables $[u_0, ..., u_k]$. Let M be a $(k+1) \times (k+1)$ difference bound matrix where the following conditions hold:

- For every $0 \leq i, j \leq k$, the entry M_{ij} contains an upper bound $(a, b) \in \mathbb{Z} \times \mathbb{B}$ if and only if $u_i - u_j \prec a$ where $\prec$ is $<$ if b is true and $\leq$ otherwise. Furthermore M_{ij} contains ∞ if and only $u_i - u_j$ is unbounded.
- Each entry is taken from the so-called *bounds-domain* $\mathbb{D}$ which is defined as $(\mathbb{Z} \times \mathbb{B}) \cup \{\infty\}$.
- The variable u_0 is a special case that always has the value 0, thus M_{i0} denotes the upper bound of variable u_i and M_{0i} denotes its lower bound.

Now we need to define the order of bounds to define the notion of a stronger bound. Let (a, b) and (c, d) be two upper bounds then $(a, b) < (c, d)$ if and only if $a < b$ or $a \approx b \wedge \neg c \wedge d$. The special case $(a, b) < \infty$ is always true and both $\infty < \infty$ and $\infty < (a, b)$ are always false. In the following example let $\top$ denote true and $\bot$ false.

Example 13. The zone constraints of example 12 can now be written as difference bound matrix:

	0	x	y	z
0	$(0, \bot)$	∞	∞	∞
x	$(0, \bot)$	$(0, \bot)$	∞	∞
y	$(0, \bot)$	$(2, \top)$	$(0, \bot)$	∞
z	$(0, \bot)$	∞	$(3, \bot)$	$(0, \bot)$

In this example the bound $x - z$ can be derived from the sum of the bounds for $x - y$ and $y - z$ can be reduced to y. This transitive reasoning can also be applied to all possible combinations. For a given zone there are generally an infinite number of difference bound matrices possible to represent that zone. However, if we apply this strengthening of bounds to all bounds universally we obtain a canonical representation. This can be convenient for state space exploration as this representation means that each zone is uniquely represented.

Definition 48. A difference bound matrix D is called canonical if and only if there exists no difference bound matrix D' where any constraint is stronger $D'_{ij} < D_{ij}$ such that represented zone Z is the same for both D and D'.

Computing the canonical different bound matrix of any given difference bound matrix can be done by re-purposing Floyd-Warshall shortest path all pairs algorithm. Consider difference bound matrix as an adjacency matrix for a graph such that the transition weights are given by the bound. Then the shortest path algorithm minimizes each bound, which results in a canonical matrix.

Example 14. The following difference bound matrix is the canonical form of example 13.

	0	x	y	z
0	$(0, \bot)$	$(2, \top)$	$(3, \bot)$	∞
x	$(0, \bot)$	$(0, \bot)$	$(3, \bot)$	∞
y	$(0, \bot)$	$(2, \top)$	$(0, \bot)$	∞
z	$(0, \bot)$	$(5, \bot)$	$(3, \bot)$	$(0, \bot)$

However, the running time for this algorithm is $O(k^3)$, where k is the number of real-valued variables, which can be quite expensive. Therefore, an alternative way is to make sure that every operation on the difference bound matrices preserves the canonical form. Now the operations defined for zones can be defined as operations on difference bound matrices. We will assume that the input matrix D is canonical.

7.3.2 Implementation

The difference bound matrices are represented by a list of columns, where each column is a list of elements representing differences and a boolean indicating whether the difference bound matrix is empty. This empty state could also be indicated by changing one of the known bounds, for example $x - x$, from $(0, \text{false})$ to $(0, \text{true})$. The sort representing differences is defined as `Bound` which has two constructors `bound(difference : Int, strict : Bool)` and `Infinity`. The first constructor represents elements from the bounds-domain $\mathbb{D}$ in $\mathbb{Z} \times \mathbb{B}$ and the second constructor represents ∞. The column sort can then be defined as a list of these elements, resulting in `sort Column = List(Bound)`. The matrix of difference constraints is then represented by the sort `sort Matrix = List(Column)`. Finally, difference bound matrices are represented by the sort `DBM` with the single constructor `bound_matrix(bounds : Matrix, empty : Bool)`.

For the bounds several mappings are defined. The bounds are ordered by the $<$ relation, this is implemented by the mapping:

```
less : Bound # Bound -> Bool;
```

Its implementation is exactly the same as specified before using several cases. It is defined by the following equations:

```
(a != Infinity && b != Infinity) -> less(a, b) = difference(a) < difference(b)
    || (difference(a) == difference(b) && strict(a) && !strict(b));
(a != Infinity) ->                  less(a, Infinity) = true;
                                    less(Infinity, b) = false;
```

The bounds can also be added, i.e. for two bounds (a, b) and (c, d) the addition $(a, b) + (c, d)$ is given by $(a + b, c \vee d)$. Whenever either of the two bounds is infinity ∞ the result is ∞. This can be defined by the following equations:

```
                 add(Infinity, b) = Infinity;
                 add(a, Infinity) = Infinity;
b != Infinity -> add(a, b) = bound(difference(a) + difference(b),
    strict(a) || strict(b));
```

Bounds can also be negated with the exception of the infinity bound. This is implemented by the mapping:

```
inverse : Bound -> Bound;
```

Which is implemented by the following equation:

```
                   inverse(Infinity) = Infinity;
(a != Infinity) -> inverse(a) = bound(-difference(a), !strict(a));
```

The bounds in the difference bound matrix can be set to a different bound. First, a mapping is defined to set a bound in a column:

```
set : Column # Nat # Bound -> Column;
```

This is very similar to setting an integral bounds in a region. It is defined by the following equations:

```
            set(column, 0, newbnd) = newbnd |> tail(column);
(i > 0) -> set(bnd |> column, i, newbnd) = bnd |> set(column, prev(i), newbnd);
```

This represents a loop where each iteration where $i > 0$, which is the index that should be changed, the head is kept the same, but when i equals zero the current index is replaced and the tail of the list is kept. Next, the set mapping is defined over a matrix that calls this `set` mapping for column on the correct column.

```
set : Matrix # Nat # Nat # Bound -> Matrix;
```

Its implementation is almost the same as for columns, where the value for x is the column that should be changed. It is defined by the following equations:

```
            set(matrix, 0, y, bnd) = set(matrix . 0, y, bnd) |> tail(matrix);
(x > 0) -> set(column |> matrix, x, y, bnd)
    = column |> set(matrix, prev(x), y, bnd);
```

Finally, the set on the matrix is extended to setting an entry in the difference bound matrix, which is defined by the mapping:

```
set : DBM # Nat # Nat # Bound -> DBM;
```

This is then implemented by using the `set` operation on the underlying matrix.

```
(x != y) -> set(dbm, x, y, bnd)
    = bound_matrix(set(bounds(dbm), x, y, bnd), empty(dbm));
(x == y) -> set(dbm, x, y, bnd) = dbm;
```

Note that changing the bounds where $x = y$ is not allowed so in that case the original value is kept. Now the functions on zones that are present in the zoned equation system are specified as a number of mappings and a number of equations defining them in the mCRL2 data specification language.

Time elapse

The time-elapse function is defined by the mapping `time_elapse : DBM -> DBM`. For the time-elapse function every upper bound in the difference bound matrix on a variable becomes unbounded. More formally, for every $0 < i \leq k$ the bound D_{i0} is set to ∞, where k is the size of the matrix. Note that in the case of a difference bound matrix the variable indices are incremented by one, because zero is used to represent the constant zero. The result of this operation is a canonical matrix as no bound can be improved by taking a path via a weight ∞. Its implementation depends on the mapping `time_elapse_for_i : DBM # Nat -> DBM` that changes the bound for each column. This mapping is defined by the following equations:

```
           time_elapse_for_i(dbm, 0) = dbm;
(i > 0) -> time_elapse_for_i(dbm, i)
               = time_elapse_for_i(set(dbm, i, 0, Infinity), prev(i));
```

These equations represent a loop where for every iteration the last element in the column is changed to unbounded. Then the equation for the `time_elapse` mapping is defined as follows:

```
time_elapse(dbm) = time_elapse_for_i(dbm, prev(#bounds(dbm)))
```

Conjunction

The conjunction function with an expression $x - y \prec m$ is solved in several different steps. First of all the difference bound matrix must be marked empty whenever $D_{yx} < (-m, \prec)$, when the matrix is empty do not update the bounds in D. Otherwise, the bound D_{xy} is set to $(m, \prec)$ if it is stronger. If the bound D_{xy} is updated then the matrix must also be normalized again. Instead of running the full Floyd-Warshall shortest path algorithm it is also possible to normalize the matrix in $O(k^2)$, where k is the number of real-valued variables, by considering that only a single weight has changed.

The following mapping is defined that only changes a bound when it is stronger than the current bound at that position.

```
strengthen : DBM # Nat # Nat # Bound -> DBM;
```

This mapping is implemented by setting a different bound when it is stronger, as defined by the less mapping, and has no effect otherwise. This mapping is defined by the following equations:

```
(!less(bnd, get(dbm, x, y)) || x == y)
    -> strengthen(dbm, x, y, bnd) = dbm;
(less(bnd, get(dbm, x, y)) && x != y)
    -> strengthen(dbm, x, y, bnd) = set(dbm, x, y, bnd);
```

Where x, y represent a location in the matrix and `bnd` the new bound. Then the following mapping is defined that is used to decide whether to strengthen a bound or mark the difference bound matrix as empty.

```
update : DBM # Nat # Nat # Bound -> DBM;
```

This function marks the difference bound matrix M as empty whenever a conjunction operator updates an entry M_{ij} with bound $(m, \prec)$ such that M_{ji} becomes smaller than $(-m, \prec)$. Otherwise the strengthen mapping is applied to the same parameters. This can be defined by the following equations:

```
(less(get(dbm, y, x), inverse(bnd)) && x != y)  ->
    update(dbm, x, y, bnd) = bound_matrix(bounds(dbm), true);
(!less(get(dbm, y, x), inverse(bnd)) && x != y) ->
    update(dbm, x, y, bnd) = strengthen(dbm, x, y, bnd);
```

The conjunction function requires normalization. However as an optimization a partial normalization can be applied when only a single bound changes. This partial normalization is defined by the mapping:

```
partial_normalize : DBM # Nat # Nat -> DBM;
```

This function is implemented by a loop over all columns and a loop over all elements in a single column. These loops are implemented by the following mappings:

```
partial_normalize_for_x : DBM # Nat # Nat # Nat # Nat -> DBM;
partial_normalize_for_y : DBM # Nat # Nat # Nat # Nat -> DBM;
```

The mapping `partial_normalize_for_y` loops over all elements in a column and checks whether the changed bound D_{ij} improves the bound of D_{xy} to $D_{xi} + D_{ij} + D_{jy}$, which is the only new shortest possible path from x to y. This can be implemented by the following equations:

```
    partial_normalize_for_y(dbm, i, j, x, 0) = strengthen(dbm, x, 0,
        add(get(dbm, x, i), add(get(dbm, i, j), get(dbm, j, 0))));
(y > 0) -> partial_normalize_for_y(dbm, i, j, x, y) =
    partial_normalize_for_y(strengthen(dbm, x, y, add(get(dbm, x, i),
        add(get(dbm, i, j), get(dbm, j, y)))), i, j, x, prev(y));
```

Then the mapping `partial_normalize_for_x` is implemented by calling `partial_normalize_for_y` for every column in the difference bound matrix. Which is implemented by the following equations:

```
           partial_normalize_for_x(partial_normalize_for_y(dbm, i, j, x, y),
               i, j, prev(x), y);
(x > 0) -> partial_normalize_for_x(dbm, i, j, x, y) =
               partial_normalize_for_x(partial_normalize_for_y(dbm, i, j, x, y),
                    i, j, prev(x), y);
```

The mapping `partial_normalize` then calls `partial_normalize_for_x` with the correct initial parameters. It is defined by the equation:

```
partial_normalize(dbm, x, y) = partial_normalize_for_x(dbm, x, y, n, n)
    whr n = prev(#bounds(dbm)) end;
```

Finally, the conjunction can be implemented by the mapping:

```
conjunction : DBM # Nat # Nat # Bound -> DBM;
```

Which can be defined using the previously defined mapping as the following equation:

```
conjunction(dbm, x, y, bnd) = partial_normalize(update(dbm, x, y, bnd), x, y);
```

Reset

The reset function is implemented by first defining a reset for single variable. A single variable x can be reset in a difference bound matrix M by changing every bound M_{ix} to M_{i0} and M_{xi} to M_{0i}. For this purpose the following mapping is introduced.

```
reset_for_i : DBM # Nat # Nat # Nat -> DBM;
```

It is implemented by an iteration that sets bounds D_{ix} and D_{xi} at the same time. It is defined by the following equations.

```
           reset_for_i(dbm, x, 0) = set(set(dbm, 0, x, bound(-n, false)),
               x, 0, bound(n, false));
(i > 0) -> reset_for_i(dbm, x, i) = reset_for_i(
        set(set(dbm, i, x, get(dbm, i, 0)),
            x, i, get(dbm, 0, i))
        )
        , x, prev(i), n);
```

This function is then extended to a set of reset variables, given by the following mapping.

```
reset : DBM # List(Nat) -> DBM;
```

This mapping applies the `reset_for_i` mapping for every variable x in the set λ. This operation does not require normalization and can be defined by the following equations.

```
reset(dbm, []) = dbm;
reset(dbm, i |> selection) = reset(
    reset_for_i(dbm, i, prev(#bounds(dbm)), 0), selection);
```

Initialize

Similarly to regions a zone can be initialized with the only possible value being a list containing all zeroes for every real-valued variable. First, a mapping `fill_column : Nat -> Column` is defined that fills a single column with zeroes. This mapping is defined by the equations:

```
           fill_column(0) = [];
(i > 0) -> fill_column(i) = fill_column(Int2Nat(pred(i))) <| bound(0, false);
```

These equations represent a simple loop where each iteration the bound $(0, \perp)$ is added. This mapping then extended to a mapping `fill_matrix : Nat # Nat -> Matrix;` which operates on matrices. This mapping is defined by similar equations, using the `fill_column` mapping.

```
           fill_matrix(n, 0) = [];
(i > 0) -> fill_matrix(n, i) = fill_matrix(n, prev(i)) <| fill_column(n);
```

Finally, the mapping `initialize : Nat -> DBM` is defined that constructs a difference bound matrix. It is defined by the following equation:

```
initialize(n) = bound_matrix(fill_matrix(succ(n), succ(n)), false);
```

Is empty

As the update function checks whether the resulting difference bound matrix is empty it is only necessary to check whether the boolean variable `empty` of a different bound matrix representation is true or false. So the `is_notempty` function is implemented by `!empty(dbm)`.

k-normalization

This is a function that is not required for the correctness of the zoned equation system. However, it is required to ensure that the resulting boolean equation system is finite whenever all other data variables can be finitely instantiated. The idea is the same as the unbounded constraints in regions. Whenever a constraint in the difference bound matrix exceeds the upper bound it becomes unbounded.

The first mapping defined for this purpose applies the k-normalization to a single upper of lower bound and is defined as:

```
k_normalize_bound : DBM # List(Nat) # Nat # Nat -> DBM;
```

Where the list defines the list of upper bounds, and the two natural numbers the position in the difference bound matrix. Given a bound D_{xy} this is implemented by a number of cases. Let $\vec{k}$ be a list of upper bounds. If a bound exceeds the upper bound, i.e. $(\vec{k}_y, \bot) < D_{xy}$, then the new bound of D_{xy} is unbounded, which is `Infinity`. If a bound is below the lower bound $D_{xy} < (-\vec{k}_y, \bot)$ it is set to $(-\vec{k}_y, \bot)$.

Its implementation is defined by the following equations:

```
(!less(bound(klist . x, false), get(dbm, x, y))
    && !less(get(dbm, x, y), bound(-(klist . y), false))) ->
        k_normalize_bound(dbm, klist, x, y) = dbm;
less(bound(klist . x, false), get(dbm, x, y)) ->
        k_normalize_bound(dbm, klist, x, y) = set(dbm, x, y, Infinity);
less(get(dbm, x, y), bound(-(klist . y), false)) ->
        k_normalize_bound(dbm, klist, x, y) = set(dbm, x, y, bnd)
            whr bnd = bound(-(klist . y), false) end;
```

The first case is the remaining possibility besides the two other enabling conditions and keeps the difference bound matrix the same. The second equation implements the first case and the third equation the second case. Now the mapping `k_normalize_bound` is applied to every position in the matrix. This is achieved by the following two mappings:

```
k_normalize_for_x  : DBM # List(Nat) # Nat # Nat -> DBM;
k_normalize_for_y  : DBM # List(Nat) # Nat # Nat -> DBM;
```

Which take the list of upper bounds and two natural numbers for the position in the difference bound matrix. Its implementation is a loop over all entries in the difference bound matrix, where each iteration the mapping `k_normalize_bound` is applied. This is implemented by the following equations:

```
    k_normalize_for_x(dbm, klist, 0, y) = k_normalize_for_y(dbm, klist, 0, y);
(x > 0) -> k_normalize_for_x(dbm, klist, x, y) =
    k_normalize_for_x(k_normalize_for_y(dbm, klist, x, y),
        klist, prev(x), y);

    k_normalize_for_y(dbm, klist, x, 0) = k_normalize_bound(dbm, klist, x, 0);
(y > 0) -> k_normalize_for_y(dbm, klist, x, y) =
    k_normalize_for_y(k_normalize_bound(dbm, klist, x, y),
        klist, x, prev(y));
```

The first mapping loops over all columns and the second mapping over all rows in a column. The result of this function is not a canonical difference bound matrix, as such a normalization function is introduced that normalizes a difference bound matrix where every entry is potentially changed, as opposed to the partial normalization from before. For this purpose three mappings are

introduces to implement Floyd-Warshall shortest path all pairs algorithm. The first to mapping to iterate over number of vertices in the graph. This essentially computes shortest paths with one to n vertices, considering all possible paths via another vertex. The other two mappings to iterate over all possible edges:

```
normalize_for_i : DBM # Nat # Nat # Nat -> DBM;
normalize_for_x : DBM # Nat # Nat # Nat -> DBM;
normalize_for_y : DBM # Nat # Nat # Nat -> DBM;
```

The natural numbers indicate the parameters x, y, i. The first mapping iterates over every vertex i, from n to zero, and applies the `normalize_for_x` mapping. The mapping `normalize_for_x` iterate over all entries in the difference bound matrix and strengthen D_{xy} whenever the path via i is shorter, which has length $D_{xi} + D_{iy}$. These mappings are implemented by the following equations:

```
           normalize_for_i(dbm, x, y, 0) = normalize_for_x(dbm, x, y, 0);
(i > 0) -> normalize_for_i(dbm, x, y, i) =
    normalize_for_i(normalize_for_x(dbm, x, y, i), x, y, prev(i));

           normalize_for_x(dbm, 0, y, i) = normalize_for_y(dbm, 0, y, i);
(x > 0) -> normalize_for_x(dbm, x, y, i) =
    normalize_for_x(normalize_for_y(dbm, x, y, i), prev(x), y, i);

           normalize_for_y(dbm, x, 0, i) =
    strengthen(dbm, x, 0, add(get(dbm, x, i), get(dbm, i, 0)));
(y > 0) -> normalize_for_y(dbm, x, y, i) =
    normalize_for_y(strengthen(dbm, x, y, add(get(dbm, x, i), get(dbm, i, y))),
            x, prev(y), i);
```

Finally, the mapping that applies the k-normalization to a difference bound matrix can be introduced:

```
k_normalize : DBM # List(Nat) -> DBM
```

Its implementation simply uses the previously defined mapping to achieve its goal, resulting in the equation:

```
k_normalize(dbm, klist) = normalize(k_normalize_for_x(dbm, klist, n, n))
        whr n = prev(#bounds(dbm)) end
```

Now in the zoned equation system k-normalization can be used by applying the `k-normalize` mapping to the result of function T_z together with the list of upper bounds.

Chapter 8

Example

8.1 Drifting clock

Consider a drifting clock, which is a clock where each tick a slight inaccuracy $\epsilon \in \mathbb{R}_{>0}$ can occur. This system can be modelled by a timed automaton with a single clock x, a single input symbol "tick" and a single location l_0 as follows:

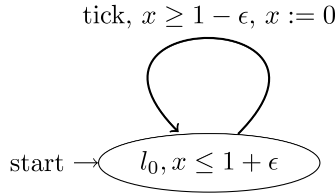

Figure 8.1: Drifting clock timed automaton

In this timed automaton the "tick" transition is enabled whenever the clock x has a value $\upsilon(x)$ greater than $1 - \epsilon$, but the invariant states that it must be smaller than $1 + \epsilon$ and it immediately resets x. The resulting timed transition system for this process would be an infinite chain of increasing time, with a transition between every two states such that the clock could tick at that time.

Now we will define several properties that could be written in the formal language called modal μ-calculus for use in the tool-set.

1. There is always a tick input possible after a sequence of ticks.
2. From the first state the tick input can be done at time α.
3. There is some sequence of tick inputs leading to a tick at time β.

Each of these properties combined with a model, or in the mCRL2 case a process, describing the drifting clock are translated into the following equation systems with real-valued parameters. The mCRL2 process specification and corresponding modal μ-formulas are presented in Appendix A.

1.
$$\begin{aligned}\nu X(l_c : L, x, T : \text{Real}) = (&\forall_{t:\text{Real}}((l_c \approx l_0 \land x + t \geq 1 - \epsilon \land x + t \leq 1 + \epsilon \\ &\quad \land T + t > T) \implies X(l_0, 0, T + t)) \\ &\land \exists_{t:\text{Real}}((l_c \approx l_0 \land x + t \geq 1 - \epsilon \land x + t \leq 1 + \epsilon \\ &\quad \land T + t > T)\end{aligned}$$

2. $$\nu X(l_c : L, x, T : \text{Real}) = \exists_{t:\text{Real}}(l_c \approx l_0 \wedge T + t \approx \alpha \wedge x + t \geq 1 - \epsilon \wedge x + t \leq 1 + \epsilon \wedge T + t > T)$$

3. $$\mu X(l_c : L, x, T : \text{Real}) = (\exists_{t:\text{Real}}(l_c \approx l_0 \wedge x + t \geq 1 - \epsilon \wedge x + t \leq 1 + \epsilon \wedge T + t > T) \wedge \tilde{X}(l_0, 0, T + t)) \vee \exists_{t:\text{Real}}(l_c \approx l_0 \wedge T + t \approx \beta \wedge x + t \geq 1 - \epsilon \wedge x + t \leq 1 + \epsilon \wedge T + t > T)$$

The sort L represents a finite set $\mathbb{L}$ that consists of location l_0 only. In the mCRL2 specification this sort is represented by natural numbers, where only the value 0 is used. For ϵ we choose the value $\frac{1}{3}$ and multiply all constants by three to obtain natural number bounds, resulting in 2 as lower and 4 as upper bound. For the constant α we will choose three and for the constant β we will choose five.

Although the translation itself is not presented, there are some things to note about the result. The parameters present are the ones occurring in the states of the timed transition system, i.e. the location l, the clock x and the last action parameter T. Furthermore, in each equation a new condition $T + t > T$ is introduced which is related to the *progress* condition.

The binding fixed point operator for each equation system is dependent on the type of property encoded. This depends on the modal μ-formula that describes this property. Typically the largest fixed point operator corresponds to infinite behaviour, as shown in the first example, and the least fixed point operator to finite behaviour, as shown in the third example. In the second example the equation does not depend on any predicate variable and as such the binding fixed point can be chosen arbitrarily. Every other constraint is derived from the enabling conditions and invariants in the timed automata.

The model check problem that we are trying to solve it always starts in location l_0 and with every real variable being zero. Which corresponds to the equation X with initial parameters l_0 and every real variable equal to zero. Now each equation is translated using both the region and zone translation, if possible, that uses the previously described implementation to solve the resulting region and zone equation systems using the `pbes2bool` tool in the mCRL2 toolset.

8.1.1 Abstracting using regions

Now we will apply the region translation defined before to each of the examples.

The first equation system is already in recursive normal form. The inequality $T + t > T$ can be replaced by the equivalent inequality $T > 0$. This only holds because T can be reset to zero in every dependency, because the parameter T is not used in any other constraint. This yields the following mCRL2 specification:

```
nu X(l : Nat, r : Region) = (forall r' : Region .
        ((is_contained(r', time_successors(r, [4, 0])) && val(l == 0)
        && satisfies(r', 0, greaterEqual(2)) && satisfies(r', 0, lessEqual(4))
        && satisfies(r', 1, greater(0))) => X(0, reset(r', [0]))))
    && (exists r' : Region . (is_contained(r', time_successors(r, [4, 0]))
        && val(l == 0) && satisfies(r', 0, greaterEqual(2))
        && satisfies(r', 0, lessEqual(4)) && satisfies(r', 1, greater(0))));
```

Note that this specification is very similar to the equation systems as defined before. The symbols `&&` and `||` are used for conjunction and disjunction respectively as typically used in programming languages. However, the function `val` is special, but it is only used to indicate that this is a boolean expression.

Solving this equation system for `X(0, initialize(2))` with the region implementation defined before results in a boolean equation system containing three equations. The solution for $X(l_0, [T = x = 0])$ is true as shown by the following proof graph:

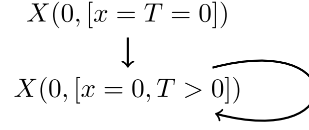

Figure 8.2: Proof graph

The second equation system fits the restrictions as well when the condition $T+t \approx \alpha$ is replaced by $T+t \geq \alpha \wedge T+t \leq \alpha$ and the condition $T+t > T$ should be replaced as well. Note that this condition is satisfied whenever $t > 0$, for this purpose we change this constraint to $T+t > 0$ and reset the variable T in every predicate variable. Note that this is only possible because no other constraint depends on the value of T. Now, the region translation can be applied, resulting in the following mCRL2 specification:

```
nu X(l : Nat, r : Region) = exists r' : Region . (is_contained(r',
    && time_successors(r, [4, 3])) && val(l == 0)
    && satisfies(r', 1, greaterEqual(3)) && satisfies(r', 1, lessEqual(3))
    && satisfies(r', 0, greaterEqual(2)) && satisfies(r', 0, lessEqual(4))
    && satisfies(r', 1, greaterEqual(0)));
```

Trying to find a solution for `X(0, initialize(2))` yields a boolean equation system consisting of a single equation. For which the solution of $X(l_0, [x = T = 0])$ yields true.

Finally, the third equation is already in standard recursive form. However, because the value of T is used in other constraints it cannot be reset to zero, so for the purpose of preserving progress another real-valued variable P is introduced. Each existential quantification is then extended with the condition $P > 0$, which is equal to a constraint $P+t > 0$ when P is always reset to zero. This change yields the following equation system:

$$\begin{aligned}\mu X(l_c : L, x, T, P : \text{Real}) = {} & (\exists_{t:\text{Real}}(l_c \approx l_0 \wedge x+t \geq 1-\epsilon \wedge x+t \leq 1+\epsilon \\ & \wedge P+t > 0) \wedge \tilde{X}(l_0, 0, T+t, 0)) \\ & \vee \exists_{t:\text{Real}}(l_c \approx l_0 \wedge T+t \approx \beta \wedge x+t \geq 1-\epsilon \\ & \wedge x+t \leq 1+\epsilon \wedge P+t > 0)\end{aligned}$$

The region translation can be applied to this equation system, yielding the following mCRL2 specification:

```
mu X(l : Nat, r : Region) = exists r' : Region . (
        is_contained(r', time_successors(r, [4, 5, 0])) && val(l == 0)
        && satisfies(r', 0, greaterEqual(2)) && satisfies(r', 0, lessEqual(4))
        && satisfies(r', 2, greater(0)) && X(0, reset(r', [0, 2])))
    ||  exists r' : Region . (is_contained(r', time_successors(r, [4, 5, 0]))
        && val(l == 0) && satisfies(r', 1, greaterEqual(5))
        && satisfies(r', 1, lessEqual(5)) && satisfies(r', 0, greaterEqual(2))
        && satisfies(r', 0, lessEqual(4)) && satisfies(r', 2, greater(0)));
```

Trying to find a solution for `X(0, initialize(3))` yields a boolean equation system with seven equations. With the result that $X(l_0, [x = T = P = 0])$ yields true.

8.1.2 Abstracting using zones

The first equation system cannot be solved using the zone abstraction because of the dependency on another predicate variable in a universal quantification. As such this example shows that some, possibly interesting properties, cannot be handled by both methods.

The second example fits the restrictions and can be translated using the zone abstraction, which results in:

```
nu X(l : Nat, z : DBM) = val(l == 0) && !empty(
    conjunction(
        conjunction(
            conjunction(
                conjunction(time_elapse(2), 0, 1, bound(-2, false)),
            1, 0, bound(4, false)),
        0, 2, bound(-3, false)),
    2, 0, bound(3, false))
    );
```

Solving this equation system for `X(0, initialize(2))` using the zone implementation defined before yields a boolean equation system with two equations and the solution for $X_{0,\{(0,0)\}}$ is true.

For the third equation system the same changes are made as for the region translation, after which the zone translation can be applied. However, for the purpose of preventing duplication, which would mean that the term has to be rewritten twice, the zone resulting from T_z is created once and passed as an argument to a different equation, which only checks for non-emptiness and has a dependency on another predicate variable when necessary. This is an optimization that could also be made in the translation or as part of a different translation step.

This results in the following equation system:

```
mu X(l : Nat, z : DBM) = X_Post2(
    conjunction(
        conjunction(
            conjunction(
                conjunction(
                    conjunction(time_elapse(z), 0, 1, bound(-2, false)),
                1, 0, bound(4, false)),
            0, 2, bound(-5, false)),
        2, 0, bound(5, false)),
    0, 3, bound(0, true)))
    || X_Post1(
    k_normalize(
        conjunction(
            conjunction(
                conjunction(time_elapse(z), 2, 0, bound(3, false)),
            0, 2, bound(-1, false)),
        0, 3, bound(0, true)),
    [4, 5, 0]));
mu X_Post1(z : DBM) = !empty(z) && X(0, reset(z, [1, 3]));
mu X_Post2(z : DBM) = !empty(z);
```

Finding the solution for `X(0, initialize(3))` requires a boolean equation system consisting of nine equations, yielding the solution for $X_{0,\{(0,0,0)\}}$ being true.

8.1.3 Time comparison

The last example can be scaled by increasing the time at which the last transition should take places, which is set by the constant β. For a number of runs with equal solution the time it takes to instantiate the boolean equation and solve it was measured using the `time` utility.

Every solution is obtained by using the `pbes2bool` tool of the mCRL2 toolset. The version of toolset used is revision 15138 of the trunk of the SVN repository. This version was compiled using the Visual C++ compiler version 19.12.25834, included with the Visual Studio 17 version 15.5.3

release, with "release" target optimizations enabled. The hardware specifications for the machine used for time comparison are as follows:

- Processor: Intel Core i7-2670QM.
- Memory: 24gb DDR3 1333Hz.
- Operating system: Windows 10 build 16299.

The `timeout` utility was used to specify as maximum running time of 1200 seconds, equal to 20 minutes. The exact command executed for every run is:

```
time timeout 1200 pbes2bool -itext <input>
```

For every run the size of the resulting boolean equation system is presented, which was obtained by the following command:

```
pbes2bes -itext <input> | besinfo
```

Every timing result is the average of three test runs. The time and number of equations for runs with a different value for β presented in Figure 8.3.

β	Region Time (Sec)	Region BES (#)	Zone Time (Sec)	Zone BES (#)
4	0.84	7	2.08	11
40	3.10	79	6.50	57
400	100.36	799	52.76	507
800	535.71	1599	113.53	1007
1200	965.02	2399	171.94	1507
2000	$\geq$ 1200	-	280.36	2507
4000	$\geq$ 1200	-	523.43	5007

Figure 8.3: Results of region and zone comparison

The missing region case where β equals 2000 and 4000 exceeded a timeout of 20 minutes after which it was decided to abort the test. This comparison can also be presented as a plot, as shown in Figure 8.4, to show the different growth rates. The blue line represents the time for the region equation system and the red line represents the time for the zone equation system.

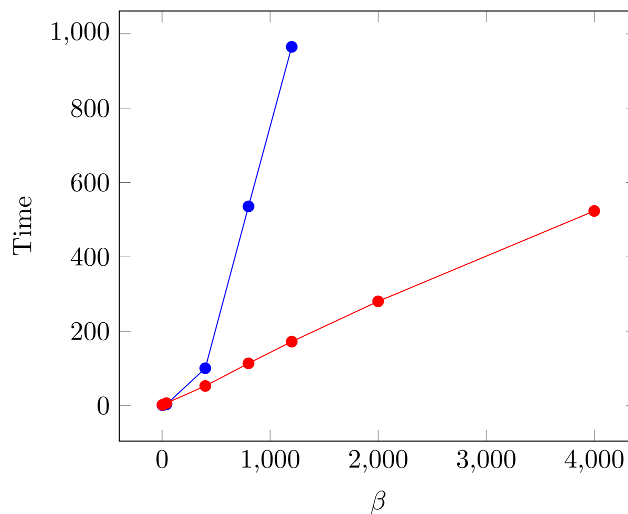

Figure 8.4: Plot of time comparison

These observations show that for a small number of equations the zone equation system took more time to solve then the region equation system. However, for bigger values for β the zone abstraction scales much better. In this case it seems that the time it takes to solve the region equation system grows quadratic in β and the time required to solve the zone equation system grows linearly. The number of equations grow linearly in both cases.

After measuring the time for instantiating the boolean equation system without solving it can be established that the majority of time is spent with instantiating the boolean equation system and solving step itself is very fast. The memory usage for most cases was quite low, less than 30MB was used.

Chapter 9

Conclusion

In this thesis we have presented two different abstractions for equation systems with real-valued parameters. Each method uses a different construct, i.e., regions or zones, to finitely partition the real-valued domain. For both methods we have shown a proof of correctness that shows that both methods offer a suitable abstraction. However the zone abstraction is more limited in the type of equation systems that it can handle. The proof of correctness for the region translation was obtained by showing a so-called consistent correlation. The proof of correctness for the zone translation was obtained by a proof graph construction.

For both methods a suitable representation in the mCRL2 data specification language was defined and implemented. Together with the implementation for every function defined as part of the translation this allowed us to automatically verify region and zone equation systems obtained from the translation. Both implementations were tested on a simple example to compare their efficiency in practice. The implementation has to be tested on more (practical) examples before it is possible to establish a definitive conclusion. However, the initial example is in support of the observation for timed automata that the zone abstraction should be better suited in practice.

Chapter 10

Future work

There are multiple interesting research opportunities. Some research questions that were noted in this thesis are presented again for completeness and some opportunities are highlighted that apply to a different domain, specifically timed automata and mCRL2.

The equation systems with real-valued parameters are the result of a translation of models and properties with time. An interesting opportunity would be to figure out which restrictions on the model and property language are required such that the result is an equation system that can be solved with the region, respectively zone, translations. This mainly relates to the usability of the tool-set in that the user does not have to know anything about the resulting equation systems.

Related to this would be extending the tool-set with a tool, or possibly added to another tool, that can apply the region and zone translation automatically. This way the abstractions can be made useful in practice, because manually defining these equation systems can be cumbersome.

It would also be interesting to further optimize the region and zone implementations. As noted in the drifting clock example the translation could be changed such that it prevents rewriting the zone twice in the implementation. The implementation could possibly be improved by looking at more efficient data-structures to represent regions and zones. For the difference bound matrix this includes a sparse representation, which overall should be more space efficient, reducing the amount of memory required to store the underlying states. For these improvements it is also important to apply the region and zone translations to real world examples to measure their potential speed-up and memory savings in practice.

One addition specifically for the restrictions on the equation systems on which the zone translation can be applied would be to extend the grammar to allow universal quantifications over data elements that are not real-valued. In this case a possible solution might be to preserve the zone. However, this extension means that the corresponding proof graph no longer has a single dependency for every vertex, as such the current proof has to be changed.

Another possible research opportunity would be to include the same extensions as presented for timed automata, but applied to the equation systems with real-valued parameters. This includes the notion of diagonal constraints, which are constraints where the difference of two clocks is compared with an integral constant. This type of constraints do not increase the expressiveness of the language that the timed automata represent, but the timed automata, and as such possibly the equation systems with real-valued variables, can be written more concisely. This also relates to the constraints for the model and property languages and possibly makes the resulting equation systems easier to solve. This extension could either be handled by extending the translation or by defining a translation from an equation system with diagonal constraints to one without.

Another extension is to change the type of updates allowed for the real-valued parameters in the equation system, which are currently quite strict. In the paper [5] there are several alternative types of updates considered for timed automata that still have decidable model checking results. This decidability also means that these types of updates could also be used in practice depending on the time it takes to solve the model checking questions. Perhaps these updates could also be introduced for the real-valued equation systems and their decidability can be explored.

Bibliography

[1] Rajeev Alur. *Timed Automata*, pages 233–264. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000. 4, 14, 19

[2] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, April 1994. 4, 17, 18, 20

[3] Johan Bengtsson. *Clocks, dbms and states in timed systems.* Citeseer, 2002. 51

[4] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer–Verlag, October 1995. 51

[5] Patricia Bouyer, Catherine Dufourd, Emmanuel Fleury, and Antoine Petit. Updatable timed automata. *Theoretical Computer Science*, 321(2):291 – 345, 2004. 66

[6] Sjoerd Cranen, Jan F. Groote, Jeroen J. A. Keiren, Frank P. M. Stappers, Erik P. de Vink, Wieger Wesselink, and Tim A. C. Willemse. *An Overview of the mCRL2 Toolset and Its Recent Advances*, pages 199–213. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. 4

[7] Sjoerd Cranen, Bas Luttik, and Tim A. C. Willemse. *Proof Graphs for Parameterised Boolean Equation Systems*, pages 470–484. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. 10

[8] Jan F. Groote and Mohammad Reza Mousavi. *Modeling and Analysis of Communicating Systems.* The MIT Press, 2014. 4, 15

[9] Jan F. Groote and T. A. C. Willemse. Model-checking processes with data. *Sci. Comput. Program.*, 56(3):251–273, May 2005. 9, 43

[10] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5(2):285–309, 1955. 7

[11] G. S. Tseitin. *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. 1983. 21

[12] Tim A. C. Willemse. Consistent correlations for parameterised boolean equation systems with applications in correctness proofs for manipulations. In Paul Gastin and François Laroussinie, editors, *CONCUR 2010 - Concurrency Theory*, pages 584–598, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. 25

Appendix A

Drifting clock example

The linear process specification of the drifting clock as presented in Example 8.1:

$$\text{DriftingClock}(l_c : L, T, x : \text{Real}) = \sum_{t:\text{Real}} .(l_c \approx l_0 \wedge x + t \geq 1 - \epsilon \wedge x + t \leq 1 + \epsilon \wedge T + t > T)$$
$$\rightarrow \text{tick}^{\text{c}}(T + t).\text{DriftingClock}(l_0, T + t, 0)$$

The modal μ-formulas.

1. $\nu X.[\text{tick}]X \wedge \langle\text{tick}\rangle\text{true}$, there is always a tick input possible after a sequence of ticks.
2. $\langle\text{tick}^{\text{c}}\alpha\rangle\text{true}$, states that from the first state the tick action can be done at time α.
3. $\mu X.\langle tick\rangle X \vee \langle tick^{\text{c}}\beta\rangle true$, states that there is some sequence of tick inputs leading to a tick at time β.