Eindhoven University of Technology

Department of Mathematics and Computer Science

Master’s Thesis

Design of a Proof Assistant for Process Algebras

by

P. Verduin

Supervisors: Prof. J.C.M. Baeten
Dr. E.J. Luit

Eindhoven, January 12, 2010
Prae factum

Abstract

**Background**  Process algebra is a subject in the field of formal methods, where processes are modeled by algebra. In “Process Algebra: Equational Theories of Communicating Processes” [1] this subject is explained with an emphasis on equational reasoning. So tool support for equational reasoning in the context of process algebras was desirable.

A short survey into process-algebra-related tools showed an absence of tools that support equational reasoning. This justifies the design and implementation of such a tool.

**Results**  This thesis presents an object-oriented design according to the philosophy of the “de Bruijn” criterion. Before arriving at the design, requirements were analyzed and specifications were derived. The design highlights several important design decisions such as the generalization of the function application, and encapsulation and typing via an abstract interface-structure.

This graduation project resulted in an implementation which was named Process Algebra Tool, or PAT. The tool comes with a well documented API.

**Conclusion**  Equational reasoning including structural induction and conditions is implemented, and useable, through rewriting. This implementation obeys the design choices and principles as were stated in this thesis. PAT is far from completed, several critical parts remain to be designed and implemented.
Preface

This thesis is part of my “Computer Science and Engineering” master degree graduation project. This master is part of the “Wiskunde & Informatica” faculty at the “Technische Universiteit Eindhoven”. In this graduation project I worked on the development of the part of PAT that deals with the rewriting of terms (i.e. the prover). The Process Algebra Tool is developed as a supporting platform of “Process Algebra: Equational Theories of Communicating Processes” written by J.C.M. Baeten et al. The development of PAT had already started prior to my arrival, and before I could focus on the rewriter, I collaborated with Erik Luit to expand and develop the design and implementation of the algebra package, the foundation of PAT.

I would like to thank Erik Luit as my tutor, and supervisor of PAT. His patience, open mind and critical reflections (and correctional skills) played an important part in the completion and refinement of this thesis. I would like to add that the salmon was very nice.

Also, I would like to thank Jos Baeten, for allowing me the opportunity to work on PAT in the Formal Methods group. Which brings me to the next point; gratitude is due to all F-buckets, the little social events were often much welcomed intermissions.

Additionally I would like to thank my parents, girlfriend, family, and friends for their support and tolerance to my social absence.
## Contents

**Prae factum**  
Abstract ................................................. 3  
Preface ..................................................... 5

### Table of Contents

9

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 Introduction</strong></td>
<td>11</td>
</tr>
<tr>
<td>1.1 Context</td>
<td>11</td>
</tr>
<tr>
<td>1.2 Problem Definition</td>
<td>11</td>
</tr>
<tr>
<td>1.2.1 Deliverables</td>
<td>12</td>
</tr>
<tr>
<td>1.3 Process Algebra</td>
<td>12</td>
</tr>
<tr>
<td>1.4 Document Structure</td>
<td>13</td>
</tr>
<tr>
<td><strong>2 Research</strong></td>
<td>15</td>
</tr>
<tr>
<td>2.1 Process Algebra(s)</td>
<td>15</td>
</tr>
<tr>
<td>2.2 Tools</td>
<td>16</td>
</tr>
<tr>
<td>2.2.1 Process Theory Tools</td>
<td>16</td>
</tr>
<tr>
<td>2.2.2 Mathematical Tools</td>
<td>17</td>
</tr>
<tr>
<td>2.2.3 Conclusion</td>
<td>18</td>
</tr>
<tr>
<td><strong>3 Requirements</strong></td>
<td>19</td>
</tr>
<tr>
<td>3.1 Proposed Overview</td>
<td>19</td>
</tr>
<tr>
<td>3.2 Functional Requirements</td>
<td>20</td>
</tr>
<tr>
<td>3.2.1 Algebras</td>
<td>20</td>
</tr>
<tr>
<td>3.2.2 Equations</td>
<td>20</td>
</tr>
<tr>
<td>3.2.3 Deriving</td>
<td>21</td>
</tr>
<tr>
<td>3.3 Non-Functional Requirements</td>
<td>22</td>
</tr>
<tr>
<td><strong>4 Specification</strong></td>
<td>25</td>
</tr>
<tr>
<td>4.1 Functional Specification</td>
<td>25</td>
</tr>
<tr>
<td>4.1.1 Use-Case</td>
<td>25</td>
</tr>
<tr>
<td>4.1.2 Scenarios</td>
<td>25</td>
</tr>
<tr>
<td>4.1.3 Message Sequence Charts</td>
<td>28</td>
</tr>
<tr>
<td>4.1.4 State Chart</td>
<td>32</td>
</tr>
<tr>
<td>4.2 Non-Functional Specification</td>
<td>33</td>
</tr>
<tr>
<td>4.2.1 The “De Bruijn”-criterion</td>
<td>33</td>
</tr>
<tr>
<td>4.2.2 Application Programming Interface</td>
<td>33</td>
</tr>
</tbody>
</table>
## Contents

5 Preliminary Design 35
  5.1 Requirements Interpretation 35
  5.2 Designing the Design 36
    5.2.1 Iterative Object Creation 38

6 Design 41
  6.1 Design Goals 41
  6.2 Overview 41
    6.2.1 Layering 42
    6.2.2 Process 43
  6.3 Decomposition 44
    6.3.1 Syntax Structure 45
    6.3.2 Centralized Rewriting 48
    6.3.3 Interfacing Types 48
    6.3.4 Parameter Semantics 49
    6.3.5 Axiom Conditions 49
  6.4 Boundaries of the design 50
    6.4.1 Theorems 50
    6.4.2 Recursive Specifications 51
    6.4.3 Parameter Manipulation 51

7 Subsystems 53
  7.1 AbstractTypes 53
    7.1.1 Types 53
    7.1.2 Terms 54
  7.2 Algebra 56
    7.2.1 Constructs 58
    7.2.2 Primitives 59
  7.3 Support Packages 60
    7.3.1 GUI 60
    7.3.2 Util 61
  7.4 Parser 61
  7.5 Rewriter 64
    7.5.1 Rewriting Process 64
    7.5.2 de Bruijn 68

8 Future Work 71

Bibliography 74

A Manual 75
  A.1 Installation Requirements 75
  A.2 Graphical Walkthrough 75

B RSE correction 81
Chapter 1

Introduction

This chapter introduces the context of this document and the context and type of the project to which this document belongs. A problem description of the project is given. It also gives a short introduction to Process Algebra. The introduction is concluded with an explanation of the document structure.

1.1 Context

Around the time of writing of this thesis, the book “Process Algebra: Equational Theories of Communicating Processes”[1] was published, an authoritative work of Baeten, Basten and Reniers. This work brings unity to the field of process algebras by presenting a mathematical framework, through which a wide variety of process algebras can be constructed and processes equations can be handled.

It is desirable that supporting facilities are developed. Such as an informational website, and an exercise environment. The project in which this graduation took place, is the project involving the development of the exercise environment. This project is done internally at the TU/e, and was already started prior to the start of the graduation project leading up to this thesis.

1.2 Problem Definition

The exercise environment is dubbed Process Algebra Tool or short PAT. This name was chosen as the environment is a tool, handling process algebras and their equations. The name is also sufficiently general, as the environment does not intend to prefer a specific process algebra. PAT’s main purpose is allowing the user to build an equation, and via a derivation prove the equation, within the syntactic context of a process algebra. The user should be able to work within every process algebra named in BBR09 [1]. So PAT should be able to support all these algebras. This includes well-known principles like recursion and induction.

The PAT development project is too big for a single graduation project, and the work had already been started. Therefore, it was divided into several parts; Input language, parser and equivalence checker / rewriter. The major themes involved in this graduation project where the algebra, the rewriter, their communication, and induction.
Design and implementation of the generic algebra had started already. This part of the tool was a good starting point to become familiar with the tool. A thorough knowledge of this part was also an prerequisite for designing the rewriter. The input language and input parser were not part of this graduation project, as this part of the tool requires a stable class structure representing the algebraic entities.

1.2.1 Deliverables
The following items are to be delivered at the end of the graduation phase:

- A predefined milestone version of PAT.
- Complete code documentation.
- Design document covering the design decisions.

1.3 Process Algebra
Process Algebra provides a description of processes with algebraic concepts; small examples are used as illustration.

**Algebra** Algebra is the mathematical subject in which structures of values and operators are studied. When we talk about a structure from the field of algebra, we call it “An Algebra”.

   Elementary algebra is the combination of generalized arithmetics and unknowns. These unknowns are called variables, such as $x$ and $y$. As in arithmetics, elementary algebra has values, of which the meaning remains constant like 0, 1 and 2, and operations like $+,-, \times$ and $\div$ that can be applied to the values. In elementary algebra these operators can also be applied to variables. Elementary algebra is an algebra. An algebra is a structure that combines constants, variables and operators.

   Universal algebra is the field in mathematics that studies the structures of algebras. In universal algebra the operators are generalized to functions. These functions have properties (e.g. associativity), and these properties imply the structure. A statement of a property of a specific function in a specific algebra is called an axiom. For example $x + (y + z) = (x + y) + z$ is an axiom, stating the associativity of the $+$ operator in the integers.

   A correct combination of values, variables and operators in an algebra is called a term. Stating that two terms are equal is called an equation. Equations can be proven to be correct (indeed equal), via a derivation, which results in two equal terms in the equation through the application of axioms.

**Process** A process is the behaviour of a system. A system is an “integrated whole” consisting of interacting entities, and may have input and/or output. Behaviour in a system is a combination of the observable transitions from state to state and the choices of alternative transitions made.
**Process Algebra** A process algebra models a class of processes in an algebraic way. So a process algebra is an algebra.

As values, a process algebra uses the constants in a system. One notable constant value in a process is a final state “output-state”, terminated. This is denoted with the zero-symbol “0” when the process is stuck in this final state (deadlocked), or with the one-symbol “1” when the process terminated successfully. In elementary algebras, variables represent unknown values, likewise, they represent unknown processes in a process algebra.

All the observable behaviour of a process is modeled through the use of functions. To do this, the action function is introduced and different observable behaviours are modeled by different action labels. These actions have a label, to signify what they represent, and the labels are followed with a dot (.) to signify their execution. The action function takes as input a label, say “a”, and a process, say x, and results in the process; that does “a” and afterwards results in the initial state of process x. This is written as “a.x”.

The choice is another fundamental concept of processes. The choice is also modeled through an operator with symbol +. For example “x + a.0” states the process that can choose to do either the unknown process x or an a-action, and then stop.

### 1.4 Document Structure

This thesis follows the lines of all general design documents, with the exception of the design choices made prior to the start of this graduation project. These design choices are discussed in the Preliminary Design, chapter 5.

The context and the requirements are investigated in chapters 2 resp. 3. In chapter 4, the requirements are analyzed and formalized into a specification. The design choices made in this project are discussed in 6. In chapter 7 an overview of the implementation as implied by the specification and design is given. In addition, some recommendations for future work are given in chapter 8. Appendix A.1 gives some installation notes and appendix A.2 provides a description of the graphical user interface of the tool.
Chapter 2

Research

This chapter is a recapitulation of the theoretical preparation taken.

2.1 Process Algebra(s)

The first step prior to the design was checking the expressivity of the architecture as present when this graduation project started. PAT is based on "Process Algebra, Equational Theories of Communication Processes" [1] (henceforward called BBR09). It was decided to do a thorough syntactical analysis on a draft version of the book, which led to the requirements and specification. The requirements and specification had, together with the architecture that was in place, a major influence on the design decisions. In this process all the SOS-rules could be discarded, as semantics have no influence on syntactic equivalence in equational reasoning.

To inventorize the complete syntax in BBR09 [1] I first sketched a taxonomy, seen in figure 2.1, of all the algebras in the book. Note that it is a rough sketch, it is used as a survey. The core algebra upon which everything else is built is MPT. The arrows relate algebra extensions. Extensions with black arrow heads indicate an increase in expressivity. Dashed lines represent embedding, where the node from which the arrow originates is embedded by the node at the end of the arrow. The grey edges and nodes are repeated extensions, they extend in a dimension already encountered. These grey parts are included as they are built upon to create new extensions.

One of the first questions this sketch raised was whether it would be possible to select a core algebra (those given in bigger balloons) and add any number of additional extensions (balloons with a “+”). This seems possible for those cases where the extensions do not add expressivity. In these cases where a function \( f \) from extension A has a function \( g \) from extension B as argument, the argument \( g \) can always be rewritten in terms of the the original algebra (a.k.a. the normal form). Then the axioms from extension A can be applied to function \( f \). This is anything but trivial in those cases where one of the two extensions increases expressivity, as one needs axioms that involve functions from both extensions. For example, when combining MPT+PR with BSP, an additional axiom for the projection function is needed. This axiom PR1 states what to do when the projection encounters the 1. This means that extensions
that increase expressivity can not be arbitrarily combined. Therefore, this issue was further left alone.

During the syntactic analysis, a discrepancy was encountered between axioms RSE5 and RSE6 [1, Table 10.8], also added in appendix B as table B.1. When I started to investigate RSE6 (which I thought to be strange), I quickly found a lemma which was proven using this axiom (see lemma B.0.1), yet the application of RSE6 in the proof was different than the definition by the axiom. Reverse definition of axiom RSE6 via the proof didn’t help, as both the defined-RSE6, and the reverse-engineered-RSE6 could lead to contradictions in certain cases. The original publication [2] showed that RSE6 was incorrectly copied (see table B.2). The proof step on which the reverse-defined-RSE6 was inspired turned out to be a combination of axioms prior and after the application of the correct RSE6.

2.2 Tools

To establish the need for the PAT tool, several other tools were examined. The examined tools are mainly from the field of Process Algebra and general proving tools. This list is by no means complete; a more complete yet rather old list of process algebra related tools can be found in [12].

2.2.1 Process Theory Tools

PAM  Process algebra manipulator, developed by Huimin Lin [14], was a tool which could generate a process algebra through a specification in a meta language. Given a process algebra, PAM could reason within these algebras using equational reasoning over axioms, and recursion using induction. PAM is no
2.2 Tools

longer maintained.

**PSF**  The next tool (actually a tool-environment, as it is a collection of inter-related-tools) is PSF, PSF stands for *Process Specification Formalism* [8]. This tool environment enables the construction of a formal specification, based on the Process Algebra ACP with some extensions. The specifications are written in a functional way. The target audience needs to be familiar with process algebra. The current area of use is the application of PSF in the field of software engineering. The specifications specify the intended use of the software, and with a simulator one can generate traces based on some input.

**mCRL**  $\mu$CRL ([http://homepages.cwi.nl/~mcrl/](http://homepages.cwi.nl/~mcrl/)) is a process algebra language based upon ACP. mCRL2 ([http://www.mcrl2.org](http://www.mcrl2.org)) developed at the TU/e, is the successor of $\mu$CRL developed at CWI. Both $\mu$CRL and mCRL2 are quite similar to PSF in that they also are frameworks of tools which require a specification as input. $\mu$CRL and mCRL2 also use process algebra for the purpose of model checking.

**Concurrency Workbench**  The Concurrency Workbench [21] developed in Edinburgh is again a framework of tools, to handle specifications semantically with the intention of model checking. In this case the process algebra used for specification is CCS.

2.2.2 Mathematical Tools

We also have a look at general proving tools, like PVS ([http://pvs.csl.sri.com](http://pvs.csl.sri.com)) and Coq ([http://coq.inria.fr](http://coq.inria.fr)). These tools were designed to have a very general scope, namely the whole spectrum of mathematics [4]. As these tools are not focused upon process algebra, their meta language is also not related to process algebra. Both PVS and Coq are based on type theory. In contrast to the process theory tools that need a specification beforehand (described above, all except PAM), PVS and Coq have an interactive interpreter. This is due to a different goal, namely proving theorems, not model checking.

It should be noted that work is done to bridge the field of mathematical theorem proving and process algebraic model checking. For example in [5] the process of verification was similar to those as propagated by the model checking community. A specification is written in a tool understandable language and modeling the system, and that specification is compiled and checked by the tool. In this case the tool was PVS. Another example is the work by Kamsteeg [13]; He formalized a process algebra in a subset of the $\lambda$calculus. His framework is suitable for construction of more process algebras, yet the emphasize in [13] is on the correctness of the process algebras and not on the application of axioms in the process algebra.

**De Bruijn Criterion**  When examining the architecture of Coq [10] I encountered the *De Bruijn Criterion* [23]. This criterium states that to increase the trust in formal proofs (optionally done by computers) that the final correctness check should be easily checkable and therefore be small, in size, in steps, and in references.
The “De Bruijn”-Criterion was never formally stated. Several different definitions of the criterion exist. The following two quotes state two different views on the subject.

“The de Bruijn criterion states that the correctness of the mathematics in the system should be guaranteed by a small checker. Architecturally this generally means that there is a ‘proof kernel’ that all the mathematics is filtered through.” (Freek Wiedijk) [23, Par. 4.2]

“There should be a representation of the proof, that can be checked by a small reliable program. Hence, if I do not trust my tool, I can write a validator for the results myself.” (Michael Franssen) [11]

The main difference in these two interpretations is where the constraining simplicity should be implemented. In the first interpretation this constraint is put in the checker, whereas the second interpretation puts it on the representation of the proof. These are both opposite sides of the same coin; enforcing them results in a higher trust of the correctness of the proof.

2.2.3 Conclusion

This short investigation suggests that much work is done in the direction of using process algebras as specification and generating the state space via the semantics of the process algebras. On the other hand, proving statements via equational reasoning is mainly done on a more general mathematical level. Only PAM supports this in the context of process algebras, but it is no longer maintained. Therefore, there is a good reason to develop PAT.
Chapter 3

Requirements

This chapter contains an enumeration of the global requirements. First an overview is given, after which the enumeration is split up into a functional and a non-functional part.

3.1 Proposed Overview

As the purpose of the system is to support reasoning about equations, while keeping close to the pen and paper method, the functionality keeps close to the requirements of the pen and paper method as well. An algebra needs to be defined by giving its functions, sets, constants and axioms. One normally starts by stating a hypothesis in the form of an equation over two terms. After this, the terms are manipulated according to the axioms (and theorems like RSP) until they are syntactically equal.

<table>
<thead>
<tr>
<th></th>
<th>Algebra</th>
<th>Type Checker</th>
<th>Deriver</th>
<th>Equation Builder</th>
<th>Parser</th>
</tr>
</thead>
<tbody>
<tr>
<td>GUI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.1: Requirements Overview

Figure 3.1 gives a high level depiction of the basic requirements. The boxes in the middle of the diagram represent functional requirements. The program needs to understand the basic building blocks of a process algebra. The tool must be able to state the equational hypothesis. Therefore, the user must be able to build the equations with the tool. PAT must be able to check these equations on syntactic correctness. After this, the user should have the tools of the equational-reasoning method to calculate a derivation as proof of the hypothesis. In other words, PAT should be a proof-assistant not a proof-generator;
a user lacking mathematical skills, stating a contradictory hypothesis, or using
an algebra that is either lacking axioms or is otherwise incomplete / unsound,
does not have to succeed in finding a proof. An interface and parser to input
the process algebras from BBR09 [1] (and optionally to modify or extend these)
is also needed. The Graphical User Interface (GUI) needs to be intuitive as to
minimize the learning curve.

3.2 Functional Requirements

In figure 3.2 concisely depicts the relation between the functional requirements
sketched in section 3.1. The Algebra parser functionality is left out of this
picture as the focus is currently on other parts (see paragraph 1.1).

![Figure 3.2: Functional Process](image)

3.2.1 Algebras

**Additions** Figure 2.1 in section 2.1 shows a multitude of algebras and extensions. PAT should support all these process algebras, and it is safe to assume more extensions will follow in due time. Therefore, there should be some unambiguous input method or scheme for the algebras. Note that in the source of BBR09 all algebras are given in a fixed pattern. In the end, it should be possible to input a new (different) process algebra (or extend an existing one) with its own operators and axioms. This should not affect any of the other functionality.

**State Saving** It should be possible during equation-building and deriving to save progress made. This includes the algebra used and the equation, optionally it may also include the current derivation and its context such as derivations previously proven assumed in this derivation, and assumptions which are for example needed with induction.

3.2.2 Equations

This section describes the requirements concerning the building of equations and their context.
3.2 Functional Requirements

Equation building Within the equation editor it shall be possible to construct an equation by combining atomic elements from a previously selected algebra. The equation consists of two terms, a left-hand side and a right-hand side, separated by an equals-sign. The atomic elements will commonly be operators, variables and constants. Independent of the chosen algebra, it should always be possible to add priority brackets. The syntax of these brackets may depend on the algebra.

To enable smooth equation building, the editor needs to be able to select any consecutive elements and delete the selection. Also positioning of the cursor on a location, at which an element should be able to be added, is needed. Positioning and deletion of elements are not done on a (sub)term hierarchical structure, as this structure is not yet know during equation building, and may change.

Set definitions Process algebras commonly consist of declared sets and functions of which only the signature is predefined. It should be possible in PAT to create and define sets. For example, the set \( \mathcal{V} \) of variables must be defined and the labels of actions should be put in the alphabet. Of these modified sets it should be possible to make subsets, to be used for example as parameters.

Function definitions It should be possible to define functions in those cases where they couldn’t be predefined, but can depend on predefined objects. After reading the algebra it should be possible to use (partial) instantiation of functions to construct the equation. The action function is a notable example of this requirement, as it needs an element from the alphabet as a label before it can be used. Another example is the encapsulation function which needs a binding set before usage; this set doesn’t need to be fully defined yet to be able to bind.

Standard communication function As the name already suggests, the standard communication function should be predefined and should always be accessible in all cases where an algebra with communication is used and only in these cases. After defining a port \( p \) and a data element \( d \), the three labels \( p?d, p!d \) and \( p?d \) should be added to the alphabet and the mapping element \( (p?d, p!d) \rightarrow p?d \) should be added to the communication function \( \gamma \).

Syntax check At any time it should be possible to check the current equation on syntax validity. If the syntax is invalid, the parsing error should give a hint about the cause of the invalidity. The parser should be able to access information provided by the algebra if this is necessary.

3.2.3 Deriving

Axioms applications Within PAT it should be possible to apply axioms to any given syntactically correct equation. To enable this, there should be a possibility to select an element from the equation to apply the selected axiom to, and to unify these two. This selection may be limited by the type of axiom that is selected.
Comments As the pen and paper method should be mimicked, it should be possible to add comments to the derivation under construction. PAT should automatically add the names of the axioms, as bare minimum, to the comments.

Solve top-down / bottom-up Another aspect of the pen and paper method is the ability to manipulate the equation both top-down and bottom-up until there is an exact match. This manipulation should be possible on both sides of the equation.

Step backwards It should be possible to go back one or more steps in the derivation, and start anew from the chosen location. This is needed because it is only natural to make errors and/or wrong choices, or it may be needed to optimize the derivation path.

Induction One of the most common proof methods is structural induction. Therefore this should be included for those mathematical structures (sorts) that are included in PAT. Induction is always done on a variable. For this variable it should be possible via its sort to automatically generate the base(s) and step(s) case(s) for a derivation. The base and step cases must be predefined per sort. Note that induction should only be done on sorts of which all terms are closed.

Recursion Principles Within equational reasoning it is also common practice to use non-axiomatized theorems like recursion principles. For example RSP in certain process algebras with recursion. Eventually it should be possible to use these theorems in a derivation.

3.3 Non-Functional Requirements

Verifiable core To increase the trust that a successful derivation is indeed a correct proof (within the context of the loaded algebra), it is convenient if the process that applies the axioms (and thus also disallows incorrect applications) is verifiable.

Platforms PAT should be platform independent, to enable easy distribution over the internet. Therefore all dependencies should be platform independent.

Documentation PAT must come with full documentation. This document is an important part of that documentation. The documentation should include all information needed to enable a third party to continue with the development of PAT. Thus a top level description is needed, a list of design decisions with argumentation, and an up-to-date Application Programming Interface (API).

Usability As was hinted at in the overview (3.1), it should be easy for new users to use PAT. Therefore it is advised to design the GUI where possible with the look and feel of the pen and paper method.

User-generated faults should be as clear as possible to the user. These faults include errors found by the syntaxchecker (equationparser) and by the deriver (rewriting/unification). It should only be assumed the user has knowledge about
process algebra, and algebra in general. PAT should therefore not report implementation details in the error messages visible to the user.
Chapter 4

Specification

In this chapter, the logical implications of the requirements discussed in chapter 3 are explored. The requirements are analyzed with a use-case and scenarios. After analysis they are translated into message sequence charts (MSC’s) and a state diagram. Some minor design decisions are taken and described in this chapter, while the major design decisions are described in chapters 5 and 6. Because this specification closely follows the requirements, it also retains the same split into a functional and a non-functional part.

4.1 Functional Specification

The translation of the functional requirements to a specification takes several steps. First the use cases are presented in a UML use case diagram, accompanied by the scenarios describing the use cases. This is followed by the translation of figure 3.2 to a High-level Message Sequence Chart (HMSC). The nodes of this HMSC are detailed in subsequent Message Sequence Charts. This chapter concludes with a State chart that describes the operation of the entire tool.

4.1.1 Use-Case

The diagram in fig. 4.1 relates the abstract use cases that were described in section 3.1. These use cases are “Add Process Algebra”, “Enter Equation” and “Apply Axioms”. To apply axioms one needs to have an equation to apply the axioms to, therefore the “Apply Axioms” use case includes the “Enter Equation” case. It is true that one needs an algebra to construct an equation. But from a user’s point of view, many of these algebras are already predefined, thus the use cases are not related.

The use case “Add Process Algebra”, which would need a parser, is not included in this project as it would make the assignment too large. This use case is consequently ignored in the specification and design, see section 1.1.

4.1.2 Scenarios

The two remaining use cases “Enter Equation” and “Apply Axiom” are described below in more detail by scenarios.
Figure 4.1: Use Case Diagram.

**Enter Equation**

**precondition:** At least one parsable (Process Algebra) model is present.

**trigger:** The user makes the decision which algebra will be used as the context.

**guarantee:** The ability to define an equation top-down, and to do a syntax check.

**main scenario:**

(a) Define the alphabet by adding all elements.

(b) Bind parameters to functions, if necessary.

(c) Instantiate the variables.

(d) Use the defined theory entities to construct the equation.

(e) Do a syntax check.

(f) Start the derivation, if the syntax was correct.

**Alternatives:**

(d)(1.1) Select any consecutive closed subset of the entities already placed on the canvas.

(d)(1.2) Delete the selected items.

(d)(1.2.1) Add brackets or a bracketed function around the selection.

(f)(1.1) Steps (a) to (c) can be done in any order, and one or more steps can be skipped.

(f)(1.2) It is always possible to go back to (a .. d).
Apply Axioms

precondition: There is at least one term, possibly part of an equation, and all terms are syntactically correct.

trigger: -

guarantee: The original term is manipulated according to the selected axiom.

main scenario:

(a) Select a term.
(b) Select a theory entity in that term.
(c) Select an axiom.
(d) Select a direction of the application of this axiom (implicitly selecting the top theory entity of one side of the axiom).
(e) The two selected theory entities are type checked.
(f) All placeholder variables in the (axiom-)term are matched to sub-terms in the selected term.
(g) Replace all placeholder variables in the term on the other side of the axiom with the sub-terms found.
(h) Substitute the result found into the selected term.

Alternatives:

(f)(1.1) If the type check fails go back to (a).
4.1.3 Message Sequence Charts

In this section, the specification is described using Message Sequence Charts. Message Sequence Charts relate static objects (depicted as columns) through their messages (depicted as arrows) and optional internal states (depicted as boxes), and thereby visualizes their dynamics. High-level Message Sequence Charts relate Message Sequence Charts in a state chart like manner. For further explanation and an in depth discussion see [16, Chap. 6].

The High-level Message Sequence Chart (HMSC) in figure 4.2 is derived from figure 3.2 and the scenarios described in section 4.1.2. The nodes in the HMSC correspond to the arrows in figure 3.2. Several relations between the Message Sequence Chart (MSC) are defined in the alternatives part of the “Enter Equation” scenario.

Note that the application of recursion principles is left out of this HMSC (see the requirements section 3.2.3). This could be added in parallel with “Equivalence Check”, optionally leading to the end state. More on this can be found in section 6.4.

Figure 4.2: HMSC describing the relation between all the MSCs in PAT.

The first MSC described in the HMSC 4.2 is “Select Algebra”. This MSC is depicted in figure 4.3. It describes how a user starts by selecting an algebra, and how this directly affects the state of PAT. Note that the SetCurrent message sets a global variable to reference the loaded algebra. This is needed as many parts of PAT must be able to access the currently loaded algebra.
The node “Build Equation” in the HMS of figure 4.2 is actually a collection of tightly connected MSCs running in parallel. For a detailed description of this node, see section 5.2.1.

There are two things to note in the MSC in figure 4.4 that describes the “Syntax Check” node from figure 4.2. The first thing one notices is that most of the complexity, the parsing itself, is abstracted away. This is done as the requirements do not tell how to parse, nor can it be straightforwardly derived. This is discussed in detail in section 7.4. The second thing to note here is that there are actually two MSCs, one for succeeding as is depicted, and one for failing. This last one simply returns “Fail” instead of “Success” and disables the proof-button if it was enabled.

Figure 4.3: MSC depicting the process of selecting an algebra.

Figure 4.4: MSC describing the process of syntax checking.
Figure 4.5 depicts the process of starting an induction proof. After the user decides to do induction the user is presented with a list of all variables from the equation built. This list contains every variable once, of which one is chosen. After the type information is retrieved, the cases associated with that type are constructed. Every base and step case gets its own derivation frame, in which the case can be proven.

The MSC in figure 4.6 is mainly derived from the “Apply Axiom” scenario in section 4.1.2. The MSC nicely depicts the absence of order in the first four messages (dashed lines), with the exception that “Select Eq side” precedes “Select Eq element”. However, it does not depict that every one of these four messages can occur an unbounded number of times, because in practice, the user can change the selections an arbitrary number of times. The messages shown in figure 4.6 represent the choices last made by the user. After the user decides it is time to rewrite, the GUI uses the messages to call the rewriter. As with the syntax check MSC of figure 4.4, the actual application of the axiom is abstracted from. See section 7.5.1 for a detailed description of the rewriting process.

Figure 4.7 depicts the equivalence check. In the HMSC (figure 4.2) the equivalence check leads to the end node. If a derivation passes the check, the user is finished. Again this MSC abstracts from the actual equivalence check, which is described in section 7.5. The abstraction is depicted as a gray block in fig. 4.7). The $E_1$ and $E_2$ in the message $Check(E_1,E_2)$ represent the derivation’s left- and right-hand side Terms, which are the terms needed to be checked on equality.
4.1 Functional Specification

**Figure 4.6: MSC depicting how to apply an axiom to an equation.**

**Figure 4.7: MSC depicting the equivalence check.**
4.1.4 State Chart

A State Chart shows all possible states of a modeled system, and relates these states with state changes, depicted as arrows. States with sub states also have a starting symbol (black dot), to indicate in which of these sub states the state starts. Parallel behaviour in a state is indicated with a dashed line separating the two parallel sub state charts. For further explanation and an in-depth discussion see [16, Chap. 7].

The state chart (figure 4.8) can be seen as a formalized refinement of figure 4.2. It adheres to the general flow as defined in the informal picture of figure 3.2. This is extended with the state information gathered from the HMSC, and some states encountered in the MSCs.

Most notable are the states implied by the MSC in figure 4.6. These states are depicted in the equation part of the editor in the prover. Another example is the “enable(proofButton)” message in figure 4.4 resulting in the safe and unsafe state of the equation builder. Note that whenever the equation is changed (the “add to the equation” transition) the builder restarts in the unsafe state.

Figure 4.8: State Chart of PAT.
4.2 Non-Functional Specification

The two transitions from “select Algebra” to the Builder represent the alternative cases where an algebra is selected and where a previously saved state is reloaded.

Several remarks need to be made with regard to the Editor. The Axioms and Comments sub-states represent data, which can change, and therefore have a state space, but these are far too large to represent. The “H” symbol in the Direction-substate implies that resets of the editor state have no influence on the Direction state. In the Equation sub-state all the arrows leading to either the Left- or Right-Side states are represented in the MSC of figure 4.6 by “Select Eq side”. Both arrows leading to the “Selected Elem” states are represented in the MSC of figure 4.6 by “Select Eq Element”. If these messages are called S and E respectively, then the LTL formula \((S \Rightarrow \Diamond E) \land (E \Rightarrow \Diamond S)\) describes their relation. Stating informally, if S holds, E holds eventually, and before every E there was an S in the past. LTL stands for linear time temporal logic, see [9] for an in-depth discussion.

4.2 Non-Functional Specification

This section is considerably smaller than the functional specification. This is due to the fact that it is far easier to formalize the implication of the functional requirements than it is for the nonfunctional ones.

4.2.1 The “De Bruijn”-criterion

To increase the trust in the derivations of PAT, it was decided that the design should uphold the De Bruijn criterion. Both interpretations of the criterion, as stated in section 2.2.2, will be accomplished by the design.

4.2.2 Application Programming Interface

To supply a general and consistent code documentation, Javadoc is used (Javadoc is supplied with Java). The major advantage is that with Javadoc one can check the completeness of the documentation regarding all inputs and outputs of all classes and methods. Javadoc can also generate an HTML document containing the API. The API can be considered to be a bridge between this document and the actual code.
Chapter 5

Preliminary Design

As described in section 1.1, there already was a conceptual design when I arrived. This implies that several design decision were already made. It is important to distinguish the design decisions that I made, or at least played a major part in, from those on which I did not have any influence. In this chapter I will identify and clarify the most important of the design decisions made prior to my arrival.

5.1 Requirements Interpretation

Paradigm  One of the first and important design decisions one can make during a software engineering track, is the choice of paradigm in combination with the programming language. A purely object-oriented design in combination with Java was chosen. To enhance the understandability, only the aggregation, generalization and the dependency relations were used in the design. Java was chosen because of its portability (Operating System Independence). Additionally, it also provides solid encapsulation to enforce layering, and a code documentation standard.

Algebra Input method  It is envisioned that the construction of the supported algebras will be done through parsing the \LaTeX{} source files that describe the theory as taken from the \LaTeX{} source files of [1]. Although this would be ideal, generating a complete algebra from the \LaTeX{} source files is probably too complex. Other options are to be researched (see section 8).

The main reason to use \LaTeX{} is because many characters in BBR09 [1] do not have a unicode equivalent. Therefore all symbols in PAT will be internally constructed via \LaTeX{}. This is a more flexible method than, e.g., maintaining a translation table. Also, since many symbols in [1] do not have a unicode equivalent, the connection with the book is better.

Algebra genericity  The above design decision already implies some genericity with respect to the algebra that can be used and restricts the possible assumptions. In other words, hard coding the theories is not an option. It was decided to embrace this, and attempt to make the tool as generic as possible with respect to process algebras in general. An overview of different process algebras
5 Preliminary Design

can be found in [3]. Yet to minimize the scope, some process-algebraic specific assumption were made. The most important ones are described in section 5.2.

5.2 Designing the Design

Figure 5.1 depicts the UML class diagram of the algebra package from the time I started. The GUI package is left out of scope here, as it did not incorporate any significant design decisions. This diagram in figure 5.1 only displays those algebra-related classes of which it was certain that they were needed. The actual diagram contained more (unrelated) classes; these classes have either been eliminated from the design or have been redefined in which case they are discussed in Chapter 6. Section 7.2 discusses the current state of this class diagram. The actual design also included more relations, for example “Theory” had aggregate relations to almost all other classes which have been left out for clarity.

Most of the classes and their relations as seen in figure 5.1 are a logical consequence of the definition of algebra and its intended use. For example, start with a Theory, which is a collection of entities which are called TheoryEntities. Notice that a Theory can include another Theory, so Theory is also a TheoryEntity. As everything in a Theory is a TheoryEntity it follows that TheoryEntity is the (abstract) base class of algebra. Several entities that commonly reside in theories can easily be identified, like Axiom, Equation, Function and Variable. Axioms and Equations consist of terms. Since functions, variables and constants are (part of) terms, the Term interface was introduced. In addition, axioms may contain Conditions. As PAT supports process algebras the Action is also introduced, which is a special case of a function.

Although there is semantically no difference between a function and an operator, for the sake of clarity a syntactic difference is defined. Operators do not use brackets and operators can occur in prefix and in infix notation, if they have one, respectively two arguments. A consequence is that operators can never have more than two arguments. When we talk in this context about func-
5.2 Designing the Design

The ideas, we mean functions in prefix notation with brackets encapsulating their arguments. Both operator and function are implemented in the Function class. In this sense, binders like $\sum$ and $\forall$ are parameterized low priority operators.

**Constant as a separate class** It is common practice to interpret constants as functions which require no input, and always have the same output. With this definition, a constant would be a special case of a function. Yet it was chosen not to subclass `Constant`, but to give it its own class under TheoryEntity. This is mainly because constants are handled as separate concepts in BBR09 as well. Also, because the implementation of constant would only restrict, and never expand the functionality of Function, it is not logical to make Constant a subclass of Function. One could even consider to make Function a subclass of Constant, but this would be counterintuitive, and in a purely algebraic sense the inheritance notion is incorrect.

**Labels storage** In contrast to the case with the constant, labels weren’t incorporated as a class, but rather as a mandatory argument to create an Action function. Here the label is stored inside the action, and every action is named after its label.

The set $\mathcal{A}$ is called the alphabet, as is defined in BBR09, and is used in every process algebra covered by the book. As labels need to be part of this set $\mathcal{A}$, references to the instantiated Action objects are stored in $\mathcal{A}$. To generalize $\mathcal{A}$, the class `Set` was introduced. This generalization implied an interface for the elements to be stored in the Set class; this interface is called `SetElement`. Note also that a Set may have subsets.

As a label never changes after creation, and everything that needs a reference to a label can reference the corresponding Action, there is no need for a special Label class. For an in-depth discussion on the implications of this choice see section 6.3.1.

**Term Binding** It was clear from the start that terms needed to be rewritten. In the traditional way implied by Robinson’s algorithm [19], the binding of the term match is stored in data structures managed by the algorithm. The OO paradigm implies this object property should be included in the object state; therefore there was a need for a new super class directly under TheoryEntity, which should make this binding possible for all theory entities that could possibly be bound. The class `BindableTheoryEntity` was introduced for this purpose.

It may be argued that this choice results in a violation of the de Bruijn criterion. In my opinion it does not, as it does not restrict the term rewriter (optionally written by someone else). The choice where bindings are made has no influence on the possibility to execute the type-correctness check in the rewriter. This check determines whether or not the de Bruijn criterion is met.

The class `BindableTheoryEntity` breaks the management into smaller parts. It only has to be checked for each bindable object whether the binding was already made. Therefore, it is not necessary to search the complete collection of bindings. Note that the check whether the binding is legal should still be in the rewriter, as this functionality is completely different. This subject is discussed in detail in sections 6.3.2 and 6.3.3.
Auxiliary functions  Section 5.1 concluded that the focus is on a (general) process algebra. Thus all terms are of the sort Process (P). However, there are also functions in the process algebras that are not of type P after β-reduction (argument resolution). Common examples are the functions over the labels, the communication function γ and the rename function f. These functions were classified as auxiliary functions in the class AuxFunction. An auxiliary function was defined to be stricter than a Function. A function can have any number of Terms as argument. In contrast, an auxiliary function is defined as a partial mapping Set × ... × Set ↦ Set, where the domains have a fixed order. As Set uses SetElement as base input type, every class implementing this interface could be part of the domain or the range.

Both the Set and AuxFunction classes are used in several process algebras as parameters in axioms, therefore they should be bindable. It seemed logical to subclass the naturals (and other well-known algebras) under Set. However, this introduces several problems, for example how to put variables in these sets, and how to distinguish these variables from process variables as explicit type information is not stored. This matter is discussed further in section 6.3.4.

5.2.1 Iterative Object Creation

This section describes a design decision that does not influence the Algebra package directly, but rather the usage of the algebra. More specifically, it concerns object instantiation. Theory-Entities are created in an iterative way, which at first glance may not look intuitive. There is a difference between TheoryEntities in algebras. Some entities are directly given by the algebra, e.g. constants; these (concrete) entities can directly be used in equations. Other (abstract) entities are defined by the context for which user interaction is needed, e.g. the elements of the alphabet need to be defined by the user. This is done with the EquationBuilder.

The main parts of the EquationBuilder are the abstract pane, the concrete pane and the canvas (see figure A.2). The abstract pane contains buttons for entities that are not completely defined yet, such as functions with a parameter and Sets. The concrete pane contains buttons for all entities that can be used in equations, such as actions and brackets. When a button on the concrete pane is clicked, the corresponding entity is added to the equation at the current cursor position.

Figure 5.2 depicts the creation of a parameterized function. This case was chosen because most other cases are simplifications of this case. A parameter needs to be bound to the function before the function can be used in an equation. The parameter itself does not necessarily need to be defined at this moment. The figure shows the creation of an additional function, which is more concrete than the (abstract) unbound version. If the parameter in question is a set, a button to refine the set is added to the abstract pane which is not shown in the figure. The abstract function is kept as it may be used as a starting point for another concrete function.

Another common abstract TheoryEntity is the set; the user is asked to add elements just like the user is asked for a parameter with an abstract function. Depending on the type of the elements contained in the set, a concrete TheoryEntity is created, e.g. an action is created in case of the alphabet \( \mathcal{A} \) and a variable is created in case of \( \mathcal{V} \). Unlike the functions, these TheoryEntities are
5.2 Designing the Design

not simple clones, but rather are clones of a template object owned by the set. This template object represents the type of the elements of the set.

All concrete TheoryEntities can be used in an equation. Figure 5.3 depicts the creation of a FunctionApplication in an equation. The equation is stored by the Canvas. The step in which the function application is created is special for the function case. Constants and variables skip this step, and simply pass their own reference to the canvas (to be stored in the equation), as only one instance of these entities are needed.
Chapter 6

Design

In this chapter the design is covered in a birds-eye view. All major design decisions are discussed in this chapter. The next chapter goes into more detail, and discusses most classes with explanations of their relations.

6.1 Design Goals

Algebra The genericity with respect to algebras discussed in 5.1 remains a design goal. This goal has been used several times as guidance underlaying a design decision. Most of these decisions aren’t as logically straightforward as one would expect. Often a design goal conflicts with the necessity that PAT must be able to actually manipulate concrete equations (see sections 3.2.2 and 3.2.3). This implies that whenever a generalization was made, there also had to be at least one implementation of that generalization. Or there needed to be strong arguments that several implementations would be needed in the future.

Prover Another point which steadily evolved from a requirement to a design goal is the “De Bruijn” criterion. To facilitate realization of the criterion this goal is subdivided into two subgoals, representing both views of the criterion. This is described in section 2.2.2. One subgoal is to have one single isolated “kernel” that should at least do the type check. The other subgoal is to minimize the proof representation.

6.2 Overview

The first step in a design towards a solution that satisfies the requirements, is splitting up the problem into domains. As this is already naturally done in the requirements (see figure 3.1), it is only logical to derive the domains from that division. As Java is used, the concept of packaging is henceforward mapped to problem domains. Please note again that the “Algebra Parser” is skipped, the component that should parse input files and build the algebras, as this is out of the scope of the current project. The resulting package-structure can be seen in figure 6.1.

The “Equation Builder” from figure 3.1 is merged into “GUI”, as the equation building is for a large part graphically oriented. The requirements implied
that the equation builder has some data storage behavior (equation and context) and some control behavior (entity creation and handling). The storage part of the equation building can easily be handled by the algebra. The control elements are general purpose functionality, such as exceptions, name handling and non-algebra-related storage and communication. Therefore, these parts are put together in a new package named “Util” where these parts can be accessed by all the classes in PAT. In section 7.3.2 all these individual parts are described in more detail.

The “Deriver” and “Type Checker” are also grouped together. This is done as both are too small to merit their own packages, yet are functionally too different from the other domains that they can not be included there. They also both have in common that they manipulate the algebra.

The algebra is subdivided over two dimensions. On one side there are the specializations, namely “Data” and “Constructs”, marked in figure 6.1 by the sub-package relation. On the other side there is the generalization, the “AbstractTypes” which has its own generalized “Data” and “Constructs” sub-packages, marked by a realization relation from Algebra to AbstractTypes. This division was made because the Algebra package became too monolithic. The AbstractTypes package (with its sub-packages) can be seen as a blueprint for the algebra package, as this package contains the core interfaces of the algebra.

### 6.2.1 Layering

From a user’s perspective of figure 6.1, one traverses several layers of packages and back when working with the tool. This is visualized in figure 6.2. This infor-
mal picture depicts PAT’s layered design. There is no strict layering structure:
which is due to the fact that not all layers are always used. Note that all layers
correspond to packages, except the “Prover” and the “Syntax checker”, which
are the disjunct parts of the “Deriver” package. Note also that the Abstract-
Types interfaces are marked by the thin layers between their implementations
Algebra, Data and Constructs and the packages that use the interfaces.

Figure 6.2: Layering.

6.2.2 Process

The following Petri-net (figure 6.3) describes the functionality in a formal way.
Note the similarity with figure 4.8. A petri-net is a net of interacting (arcs/arrows)
states (circles) and actions (boxes). Possible executions of petri-nets are defined
by the location of tokens (dots in a state). Petri-nets were introduced by Petri
in [18]. The arcs with double heads on one side are reset arcs, which remove
all tokens if the corresponding transition fires. However, they are never an ob-
struction to fire. The first transition in the petri-net is always the choice of the
algebra one wants to work with. In figure 6.3 the dots above and under the
first transition (and the extra arrows leading out of the starting state, and the
arrows leading into the following state) represent the different algebras the user
can choose from.

After the choice of algebra, the user arrives in the state where the user can
construct the equation from elements of the algebra. In the following, these
algebra-elements are called tokens which are generated by the “Add to Equa-
tion” transition. The syntax checker accepts the string of tokens built by the
user, parses these and via construction of an “Equation” (from the Constructs
package), it checks the syntactical validity. The syntax check has as precondi-
tion for successful completion that the string of tokens may not be empty. One
can only start deriving a proof after a syntax check. If the equality check passes,
the process is completed.

To add recursion to this model, one could imagine a coloured token gener-
ator just after the theory. The colours are needed as filters for the reset arcs.
One only would need to check that the specification satisfies the “Recursive
Specification Definition” of BBR09 [1, Def. 5.2.1] when a proof is started.
Induction

In case the type of derivation is induction, the “Prove” process is not an atomic transition, but rather a small process. In this case, all variables are listed for the user, who may select one. Depending on the sort (algebra) of the selected variable, there are different induction cases. Cases are stored in the Theory or PrimitiveType classes respectively. Every case results in a separate token in figure 6.3 (every token owns its own derivation frame). In every derivation, the selected variable is substituted by its “case”. The induction requirement is stated in section 3.2.3; section 7.2.1 discusses the internals.

6.3 Decomposition

This section describes the decomposition in more detail (see figure 6.4). If one looks at figure 6.3, one can identify the central nodes with the states in figure 3.2; namely the starting node, the node after BCP and the node after Prove. These therefore have their own GUI element, respectively PAT, EquationFrame and DeriveFrame, as shown in figure 6.4. Note that there is a specialized GUI for recursive specifications, holding a reference to EquationFrame to build every individual recursive equation. See section 7.3.1 for an explanation of the classes in the GUI package.

The “Type Checker” functionality is split up into an “Equation Parser” and an “Equivalence Check”. The name of the equation parser in the implementation is “List2TreeParser”. This name stems from the fact that the parser takes a list of (algebraic elements) tokens and constructs a syntax tree. See section 7.4 for an in-depth discussion.

The process of a derivation step through the application of an axiom on an equation is a combination of rewriting and equivalence checks. Therefore, the second part in the deriver package is the “Rewriter”, which does the equivalence check.
6.3.1 Syntax Structure

Because the tool is developed according to the object-oriented paradigm, the list passed to the parser is actually a list with references to objects. This gives the nice property that the resulting tree actually is a Directed Acyclic Graph (DAG) if all references to equal tokens are the same and the syntax is correct. As explained in section 5.2.1, a concrete object returns its reference every time it is used. The DAG construction goes hand in hand with the syntax check and it is concluded that the list was correct if a DAG could be created. The exact process facilitating this is described in section 7.4. Note also that the non-leaf nodes have an axil that refers to a function and that several nodes may refer to the same function.

In the DAGs, a node represents a Term. As non-leaf nodes have children they are considered to be TermFunctions, a subclass of Term. This subclass has as extra functionality with respect to term that it can return the references to its children. Nodes have no references to their parent(s). The ordering of the children (edges) represents the structure of the term.

As several lists containing tokens for propositions may be constructed in the same context, they can also have shared leaves and functions. Both equation and axiom are considered to be propositions [22]; for a detailed account of their relation see section 7.2.1. So a proposition, and even a collection of propositions, can be seen as one big multi-root-DAG. An example of this is shown in figure 6.5, which represents axiom A1 \((x + y = y + x)\). Note that the L and R markers in figure 6.5 denote different function applications, that refer to the same function.
Parameters

The requirements (section 3.2.2) imply that functions may have parameters. Although not strictly necessary, this is generalized to all Terms, as this may increase the flexibility for extensions. The ActualParameter interface allows all terms to have multiple parameters. The main benefit is that the future use of parameters on terms is not restricted. Parameters are restricted not to have parameters themselves. This restriction is introduced since allowing this would introduce several difficulties, without introducing significantly more expressivity.

Functions and Nodes

In the design at the time that I started, an occurrence of a function in an equation or axiom was represented by an instance of the function with references to its argument Terms. Thus, each occurrence of the function needed its own node in the DAG. This leads to a problem with the binding of the occurrences of a function in an axiom to those in an equation.

For a function, a node in the DAG is assigned by the parser when it encounters a function token in the equation. Now consider an axiom with two action functions on the same side, both with the same label, as illustrated in figure 6.6. If this axiom is unified with an equation, the first action from the axiom (a.) that is encountered can simply bind to the corresponding action from the equation (p.). “First” here means the action encountered first in time, as no assumptions are made about the structure of the axioms or the equation. Neither are assumptions made about the unification process; consequently nothing is known about where this action is in the syntax tree. Now, further down the unification process-path the second action (a.) is encountered. Obviously, the rewriter tries to bind this action to the current token in the equation (q.) which is in principle possible if the token is also an action.

However, this is not correct in general: because both actions had the same label in the axiom, both action labels in the equation should also match exactly. Therefore, the second action in the axiom should only be matched to an action with the same label as the first action-binding in the axiom (p. in figure 6.6).

To check whether the second binding is legal, either the process should backtrack, or every action binding made should be stored in a global array. Neither options are optimal: the first takes too much time, the second takes space and time. The binding between the actions a. and p. was already made, so the
second option duplicates information already present.

An obvious solution to this problem is to have only one node per action in each axiom and in the equation. In the original design, this is not possible as the different appearances of the actions in the equation may have different arguments. Adding both arguments as child nodes to the action node creates ambiguity, since there is no way to determine which argument(s) to choose when the unification process arrives at the node unless additional administration is kept.

Another solution is to separate the label from the action class. The idea was that different actions would have a reference to a label and the label would become the bindable object instead of the action. It was decided to generalize this idea to the concept of a function application as an entity which represents the application of a function to its arguments. The situation in figure 6.6 is now transformed into the situation in figure 6.7.

The class FunctionApplication has a reference to the function of which it is an application and to the arguments of the function. The bindable entities are the actions and the parameters. A function does not need to be bound, as all occurrences in equation and axioms can be represented by the same object. Obviously, the parameters of a function do need to be bound.

The introduction of the FunctionApplication has two advantages over the original design. First, the already bloated function class is split up. Second, the action class remains unchanged which avoided extra coding needed to manage and control the new label class.

Terms in which only a particular associative function and leaf nodes occur can be evaluated in any order. Therefore, it was decided to add another abstraction, which is the introduction of two specializations of the FunctionApplication. One specialization has a fixed number of arguments, whereas the other, representing associative functions, has an arbitrary number of arguments. This allows a DAG modulo Associativity which is discussed in more detail in
sections 7.2 and 7.4.

6.3.2 Centralized Rewriting

The rewriting and equivalence check can easily be incorporated in the Term interface via the object-oriented paradigm. This could be accomplished by demanding the term to implement a specific method, for example “match()”. This would subdivide the problem into a distributed set of smaller problems, that are individually easy to solve and verify. Every match function would only need to check if the given term to match to would be of the same class, and/or check if it could bind. If the match would be successful the term would then call match on its children (outgoing edges). This method raises several objections.

Firstly, the scope suddenly includes a big part of the algebra. As the type check is done inside the object itself, it is hard for the caller to know which class it is actually calling. If one considers the possibility of extensions, this would even break the package-related limit of the scope.

Secondly, once inside the class, the implementation of the match method can easily access and change all fields of the class. Thus the state-space of the method could potentially be very large. There are almost no facilities (in Java, see section 5.1) to constrain or check this.

Both of these aspects counter the De Bruijn criteria in a similar manner. Writing one’s own rewriting implementation would require an in-depth knowledge of the algebra package. Replacing the original process, which was built into the algebra classes, would require to find and replace all the scattered match methods. Centralizing the match method into one Rewriter class (section 7.5.1) solves the first problem but not the second. The rewriter would still need information about the nodes in the DAG, and the usual way to do this object-orientedly is by calling methods on that object. This raises the question, how to contain and constrain these calls?

6.3.3 Interfacing Types

The calls from the rewriter to the DAGs (and the algebra in general) need to be contained and constrained to reduce complexity. This means that the rewriter should be able to reason about the existence of typed objects without any other knowledge of the objects.

To realize this, all communication is constrained to run through a small set of interfaces. These interfaces have a minimal number of methods, which have simple functionality and a minimal communication language. This minimizes the state-space and complexity and thus constrains each individual call. The interfaces only allow the simplest data to be communicated, preferably (Java) data primitives such as boolean, int and String. Of these three String is the only class, but this is allowed because all objects of this class are immutable. In addition, interfaces without methods are used which serve as type identifiers. It is these type identifying interfaces that enable the rewriter to reason about the existence of the objects without detailed knowledge about the classes. The interfaces (and their structure) are discussed in section 7.1.1.

Figure 5.1 introduced Term as an interface. In paragraph 6.3.1, the functional need for the Term was explained. The Term interface is used as the main interface between Algebra and Rewriter (see section 7.1.2). It is demanded
that all information about nodes is communicated through this interface, so the rewriter only needs to know these interfaces. Because the rewriter is not allowed to have any other knowledge than the preselected interfaces, the calls are contained (see section 7.5.2).

### 6.3.4 Parameter Semantics

Parameters are sets and auxiliary functions from the algebra itself, but also terms from other (parameter-)algebras such as booleans \((B)\), naturals \((N)\) and the reals \((R)\). Two important choices were made regarding how parameters are modeled (and consequently manipulated) and how the design is implemented.

**Modeling choice** One solution is to model all parameters like the Algebra was modeled. In this case the user could use the provided axioms for these parameters to manipulate them. This would be a time consuming process for the user and rewriting parameters step by step is not considered particularly interesting for the target audience. This design would also constitute additional work on the tool.

Another solution is the use of semantics, as this was already done for sets and auxiliary functions at the moment the choice needed to be made. In this case, one only needs to put effort in the design and implementation of the parameter-algebras. Therefore this option cost less time, and was chosen. For these algebras it was decided to use the Java primitive equivalents to back up the semantics.

**Design choice** Incorporating the primitive algebras into the algebra package would amount to an almost complete rework. Solving the problem of distinguishing between entities of different algebras, as stated in section 5.2, requires an extra level of abstraction, namely an additional super class above the “TheoryEntity” class. All term implementations would need to consider this type difference and supply semantics in the primitive-type cases. Given the time available this was unacceptable.

Thus it was decided to abandon the idea to merge the algebra package with the primitive algebras, and split these up entirely. The primitive algebras are implemented in the Data sub-package of Algebra, as shown in figure 6.1. The choice for a sub-package rather than a stand-alone package was made mainly because it is similar in design. However, the implementation is completely different.

This way, the algebra package remained as it was, and the rework was circumvented. Because all the elements from the primitive algebras can only be used as parameters one could consider them second-class members of PAT. This is in a sense true, as the emphasis is on the algebra that is loaded in the Algebra package and not on the standard supplied parameters.

### 6.3.5 Axiom Conditions

Axioms containing symbols with sets and (auxiliary) functions as parameters often state an additional condition. These conditions can not be resolved solely by rewriting or unification, as their resolution implies a semantic notion. One really needs to evaluate the predicate, to conclude whether the axiom may or
may not be applied. Often the evaluation of these predicates involves the execution of one or more methods of classes that represent the parameters, like sets and (auxiliary) functions. As it is the rewriter that must check whether an axiom may be applied, it is the rewriter that needs to evaluate the condition. This again constitutes a potential threat of violating the de Bruijn criterion, as the rewriter would need detailed knowledge of different parameter implementations.

The solution proposed in paragraph 6.3.2 was reused for this problem. A minimal interface was designed, which is the only interface that the rewriter uses to access objects of these classes. A consequence of using an interface dedicated to one general condition is that every type of condition needs its own class. These interfaces and their implementations, together with several container classes, were put in a separate algebra sub-package “Constructs”. This is shown at the bottom of figure 6.1. Paragraph 7.2.1 gives an in-depth discussion of this package.

Condition Semantics

An additional complication of axiom conditions is that not all conditions are simple predicates returning true or false. For example, consider the condition of axiom CM5, “if $f(a, b) = c$”. This condition does not only return true if the existence of a matching label of “c” is found, but it also binds the (meta)-variable label “c” to the matching label.

In an ideal situation the rewriter has no knowledge of the execution of the condition, it just checks the final result. Also it is preferred that the rewriter makes all the bindings. No solution was found that satisfied all these requirements. Then it was decided on a priority basis that “no algebra knowledge” was more important than “makes all bindings”. Consequently, the check was split up accordingly, as the result of the evaluation of the predicate is inherently dependent on the implementation of the semantics. This part needs to be outside the rewriter, and can be communicated through the condition interface. The check whether the binding is correct can still be done in the rewriter without the introduction of additional functionality.

6.4 Boundaries of the design

In this section the boundaries of the design in the context of this project are discussed. These are boundaries because they were investigated, some early design decisions were made, but nothing substantial has been implemented.

6.4.1 Theorems

In section 3.2.3 there is a requirement stating that it should be possible to use recursion principles. In this paragraph the requirement is generalized to the use of theorems which is a valid generalization, as recursion principles are theorems. Propositions, like equations and axioms, and derivations have a context. Such a context typically consists of the currently loaded algebra, of variables and of conditions. When a derivation is made it is not unusual to include (or note down) theorems used, in the scope of the derivation.

For this purpose, the condition interface was abstracted to a Context Interface. This Context interface has a method “use” by which the theorems can be
called. Note that currently no theorems have been implemented. Every theorem needs to have its own class because of the interface. As theorems may need to access a layer beyond what is visible to the rewriter (fig. 6.2), it is conceivable that these classes will become part of the constructs package. However this is not strictly necessary, for example, the recursion principle RSP could be implemented as a replacement equivalence check up to the difference in variables excluding the variable names.

6.4.2 Recursive Specifications

In figure 6.4 the “RecursionSpecificationFrame” was referenced. This GUI element was implemented as proof of concept. The concept was to build a collection of equations that are independently checked on syntax. After the user has completed the collection, it can be checked whether it is a recursive specification as stated in BBR09 [1, Def. 5.2.1]. If the user would want to prove the equality of two recursive specifications, the derivation screen(s) should provide the recursion principles, and possibly other tools, to prove the equality.

6.4.3 Parameter Manipulation

Section 6.3.4 discussed the semantics of parameter algebras. Deciding when to rewrite parameters, and deciding what to rewrite to, e.g. normal form, are anything but trivial. It was decided that this should completely be guided by the user. For example, if the user wants to apply an axiom such as $\pi_{n+1}(a.x) = a.\pi_n(x)$, the user has to rewrite the parameter of the projection function in the equation to the form $m + 1$, where $m$ is a PrimitiveTerm. The rewriter then checks equality/correctness. Note that this has not yet been implemented. No interface has been designed and included in the AbstractTypes package to univocally call the necessary methods. It is also unknown to me when to call these methods in the rewriting process, and in which context.
Chapter 7

Subsystems

In this chapter all subsystems are individually explained and discussed. Every class is discussed, however for a detailed description on the method-level, one is referred to the Application Programming Interface (API).

7.1 AbstractTypes

The AbstractTypes package is a collection of interfaces that define the algebra, seen as a black box with communication. The AbstractTypes package was split off from the algebra package; it was concluded the algebra package became too bloated. The package includes all, and only those, interfaces that do not (indirectly) reference any non-interface classes. Consequently all the interfaces only refer either to basic data primitives or to interfaces that adhere to this rule. One could interpret this package as the contract that specifies the obligations that the algebra package has towards the rewriter. There are two different kinds of interfaces in this package, namely the types and the terms, as already mentioned in section 6.3.3.

7.1.1 Types

As the names already implies, the type interfaces give type information. To be more precise, they are the type information. In figure 7.1 the entire type-hierarchy and the BindableEntity interface are shown. Note that none of the types defines any methods. So if one has an object which implements one of the types, and that is the only information one has, no methods of the object can be evoked via the interface.

The top of the hierarchy is formed by Sort. The Sort interface is not yet used, but was included to relate the second level, Type and PrimitiveType. It may be useful in the future when one tries to define a function over (Primitive)Types. PrimitiveType and its (grand)children are part of package Data, which is a subpackage of AbstractTypes. The structural equality between the Algebra and the Data packages, as mentioned in section 6.3.4, can clearly been seen when comparing Type and PrimitiveType interfaces and their children. Below the Type and PrimitiveType interfaces are the basic elements of an algebra, namely the function, the variable and the constant. The action is a special type as it
was identified to be a core concept of process algebras. It is a function with a label that can vary; as this label has influence on the equality, it is a special kind of function.

The PrimitiveWrapper interface is special for the primitive types. Strictly speaking this interface should not be present. It is used as a conceptual bridge between the primitiveTypes algebras and the currently loaded algebra, more on this bridge can be found in section 7.2.2.

Figure 7.1: Class Diagram of the Types in package AbstractTypes.

Unlike all the other interfaces in figure 7.1, BindableEntity does not define a type, but functionality. All Types, of which the Term can be bound to another Term by the rewriter, include the BindableEntity interface. It was included here because it does not have any direct relations to the Terms, and has include relations with types. These include relations are depicted in figure 7.1 by inheritance relations: it relates those types that are by definition bindable. A consequence of this inheritance relation is that when the rewriter identifies a (primitive)variable or label, it can by the inheritance relations automatically access the BindableEntity interface. The relation of action to BindableEntity is special here; as it is the action label that is “variable”. Thus Actions should only bind to other actions, resulting in a smaller bind scope than both Variable types, as they can bind to anything.

7.1.2 Terms

The Term-related interfaces can be seen as the executable part of the algebra package from the point of view of the rewriter. Most of these simply return a
constant information field. All these functional interfaces (except BindableEntity) are drawn in figure 7.2.

![Class Diagram of the Terms (and related) in package AbstractTypes.](image)

Context, Proposition and PropositionTerm together form the Constructs sub-package of the AbstractTypes Package. Figure 7.2 shows that a proposition may have a context. A context is composed of "Context" objects, which can be checked and/or used. At this moment only the Check() method is called, which is used to check whether the Context is true. The Use method was added to allow extensions. A Proposition has a left hand side and a right hand side, the type of these objects is on purpose undefined. PropositionTerm adds this type information, sides are defined to be Terms. Their function and their relation to the algebra’s Constructs package is further discussed in section 7.2.1.

The core of class diagram 7.2 is the Term interface. Together with TermFunction and ActualParameter these are part of the AbstractTypes package. The Term interface, following the design decision from section 6.3.3, only returns interfaces and java-primitives. The getType method returns an interface of the type “child of Type”, where Type here is the Type interface introduced in section 7.1.1. Notice that this means that each class implementing the Term interface must either implement a child of the Type interface and return self, or must own an object implementing a child of the Type interface and return that object.

Every Term can have parameters, which are returned through an array of objects that implement the ActualParameter interface. As not all nodes in a DAG have edges, a sub-interface is needed that grants access to the edges of a DAG. As this is only the case with non-leaf-nodes (functions), this Term in-
Subsystems Interface is called TermFunction. The ActualParameter interface provides typing and identification information via strings.

The PrimitiveTerm and PrimitiveOperatorApplicationTerm are both part of the Data sub-package, and both are copies of their respective Algebra equivalents. The primitive terms only indirectly implement the ActualParameter interface (via the PrimitiveWrapper type) which is discussed in more detail in section 7.2.2. As the primitive terms are parameters themselves, they do not have parameters.

7.2 Algebra

Figure 7.3 shows the algebra package together with some interfaces from the AbstractTypes package. These interfaces are shown to clarify the internal structure. In addition, the Constructs and Data sub-packages are drawn to show their relation to the Algebra package. Figure 7.3 shows the current design of the core entities of PAT as it evolved from the design shown in figure 5.1. Several classes are conceptually the same, most notably TheoryEntity and Theory, but also Constant, Variable, and to a lesser degree BindableTheoryEntity, AuxFunction and Set.

The conceptual difference between the BindableTheoryEntities Variable on the one hand, and AuxFunction and Set on the other hand, led to the introduction of the abstract class AggregateEntity. A class should implement AggregateEntity instead of BindableTheoryEntity, if the class is a BindableTheoryEntity, constitutes a collection, and where the collection is given semantics by aggregating their elements. This is unlike Variable, which is purely syntactic. Another addition is the RecVar, the class of recursion variables, that inherits from Variable. This class is currently largely empty, but was included for future use. This class should have as invariant that all recursion variables are represented by capital letters.

The Comma, Bracket and equal classes were added as they were needed for the equation building. Different Theories may use different symbols, while retaining the syntactical structure. Added initially for use in the equation builder, the classes also play a role during parsing of equations. Apart from this, these classes are syntactic sugar.

Function Application In section 6.3.1 the concept of a function application as a class is introduced. Because FunctionApplication (FA) takes the role of parent node of Function in the DAG, FA inherited the TermFunction interface when the Function class was split. As stated in section 6.3.1, FA is conceptually an aggregation of a Function and its arguments (Terms). Via the inheritance relation between Action and Function, an FA can also refer to Actions. Via the inheritance relation between Term and TermFunction, an FA can be an argument for another FA.

FA is an abstract class which defines the relations of a function application. However it does not define how the collection of argument terms are stored and handled. There are two different FA implementations, namely NAryFA and VariadicFA. The N in N-ary-FA represents the number of arguments one application of the specific function in FA needs, e.g. all infix operators have \( N = 2 \). If there are two FAs of an associative function, and one of the two FAs
is the argument of the other, then by the associativity property all orderings of the two FAs are considered to be equal (keeping the order of the arguments). Therefore the FAs can be merged as long as the order of the arguments is retained. The class VariadicFA represents merged associative FAs. Therefore VariadicFA does not have a static number of arguments. Instead, it has an invariant: \( 1 < N \leq L \land (L - 1) \mod (N - 1) = 0 \), where \( N \) is the number of arguments of one function application, and \( L \) the total number of arguments after the merge.

**Interface Implementations**  The Term interface is implemented by the classes Constant, Variable and indirectly via TermFunction by FA. The getType method of each implementation returns its specific type through the Term interface. The parameters of a Term can be the AggregateEntities AuxFunction and Set, and PrimitiveTypeExpressions.

PrimitiveTypeExpression implements, besides the ActualParameter interface, the BindableEntity interface. PrimitiveTypeExpression acts as a bridge between the Algebra and the PrimitiveTypes, and it is further discussed in section 7.2.2.

The SetElement interface was introduced during this graduation project. This interface was not added to the AbstractTypes package because it uses the Set class as parameter and return type. This property does not comply with the requirement on the interfaces in the AbstractTypes package. SetElement together with BindableEntity form the BindableSetElement interface. This interface was introduced to reduce the number of relations between classes and inter-
faces. Most classes that implement SetElement also implement BindableEntity. These classes now just implement BindableSetElement.

### 7.2.1 Constructs

The constructs package, shown in figure 7.4, was introduced to contain everything involving Axioms, Equations, and their Context. Figure 7.4 again includes several term interfaces from the AbstractTypes package for clarity. These interfaces are Context, Proposition and PropositionTerm, which is an extension of proposition. PropositionList is another interface that extends Proposition. This interface is implemented by Equation, the container class for equation building. PropositionList is used in the communication between the equation builder and the parser as a message-container. As the equation builder mostly manipulates lists of TheoryEntities, PropositionList does not implement Term. Instead, it implements lists, as the containing tokens are not yet parsed. There is an abstract class PropImp that implements the Context management defined in Proposition. Equation does not implement PropImp because of subtle implementation details. Both Axiom and EquationTerm, implementors of PropositionTerm, inherit from PropImp.

![Figure 7.4: Constructs Package.](image)

#### Conditions

Currently the only implementation of Context is Condition. Condition is an abstract class; there are three classes refining Condition. These three conditions are IsSubSetOfCond (\(\subseteq\), e.g. the condition on \(H\) in \((BSP + DH)(A)\)), IsElemOfCond (\(\in\), e.g. axiom D3) and AuxCallCond \((f(\cdot) = \cdot\), e.g. axiom CM5). All these conditions can be negated. IsSubSetOfCond takes two Sets as input. IsElemOfCond takes a SetElement and a Set. AuxCallCond takes an AuxFunction, an array of SetElement inputs for the AuxFunction, and a SetElement as expected result to compare with the actual result of the AuxFunction. Note that there is a subtle difference between the negation of
AuxCallCond \((f(\_)) \neq \_)\) and the use of axiom CM6’s condition \((f(\_)) = \text{null}\); “undefinedness in \(f\)” is defined as the absence of a mapping element in the function. More (refinement) implementations of Condition will be needed, such as “ActualParameter not free in Term” which is used in axiom CQ1 [1, Table 10.20].

**Derivations** To manage derivations, a container class “Derivation” is introduced. A Derivation has a starting equation and two DerivationSides. The DerivationSides start at their representative side of the starting equation of the derivation. From this point on, every side contains DerivationSteps. A DerivationStep records the successful application of an Axiom, the resulting Term of the axiom application and comments given by the user.

Structural Induction needs, like a derivation, a container class “TermInduction” to manage the induction process that was described in section 6.2.2. TermInduction consists of several InductionCases. Each case is a derivation with the addition of optional hypotheses. RecursiveSpec is the container class designed for the collection of equations forming a recursive specification.

### 7.2.2 Primitives

PrimitiveTypeExpression (PTE) is part of the algebra package, and was introduced in section 7.2. PTE builds a bridge between algebra and the primitive types (see figure 7.5). Of the classes in figure 7.5 PTE is the only class that implements the ActualParameter interface (not shown in fig. 7.5).

When an ActualParameter is encountered in the rewriter, the rewriter can check whether the ActualParameter is a PrimitiveTerm. By convention, the PrimitiveTerm interface returns “this” when the getType method is called. However, PTE did not implement an interface complying with the type defined to be returned by the method. Without the PrimitiveWrapper type, the PTE (which implements PrimitiveTerm) would have to return the type dynamically, and additionally implement all other (non-type) interface that the wrapped PrimitiveTerm could implement based on the dynamic type. The PrimitiveWrapper interface thus merely identifies the ActualParameter as a PrimitiveType and can be cast to the proper PrimitiveTerm for further processing.

PTE always has just one attribute, which is the actual primitive term, PTE implements the PrimitiveOperatorApplicationTerm interface. The PrimitiveWrapper type helps to distinguish PTE from other primitive functions that implements the PrimitiveOperatorApplicationTerm interface. This PrimitiveWrapper type should only be returned by PTE objects as their Type. A PTE also has a PrimitiveType representing the sort of the primitive term encapsulated by the PTE (e.g. boolean, or natural).

The (abstract) PrimitiveType class in figure 7.5 should not be confused with the PrimitiveType interface in figure 7.1. On the other hand, the PrimitiveType class does implement the PrimitiveType interface. The PrimitiveType class is refined by the Integers class. This conceptually corresponds to Theory, and the loading of an actual theory, in the context of the Algebra package. This should easily be extendable to, for example, booleans and reals. Like Theory, refinements of the abstract PrimitiveType class supports structural induction by providing cases.
Figure 7.5: Primitives Package.

When comparing the primitive package with the Algebra package, one finds that PrimitiveEntity corresponds to TheoryEntity. PrimitiveValue corresponds to Constant. Conceptually, PrimitiveOperator corresponds to Function: it just adds a restriction on the form. Only operators are allowed, see section 5.2, to decrease the complexity of these classes and their handling. PrimitiveBrackets, like Bracket in Algebra, do not implement the PrimitiveTerm interface; these were merely added as visual aids to imply structure.

7.3 Support Packages

The following packages contain classes that only play a supporting role. These packages don’t have a strict structure. These sections are here for the sake of completeness.

7.3.1 GUI

The only structure in the GUI package derives from the chronological order in the opening and closing of the screens. This leads to the effect that most screens own their successor screens. This effect was already shown in figure 6.4. From this picture one can already derive that PAT (depicted in figure A.1) owns the equation editor and the recursive specification frames. Likewise, the recursive specification frame (figure A.8) owns the equation editor and the derive frame (figure A.7). The equation editor (figure A.2) owns apart from the derive frame also the define-function (figure A.3), define-set (figure A.4), define- auxiliary-function (figure A.5), and the induction (figure A.6) frames. Lastly the induction frame owns (optionally multiple) instances of the derive frame.

Other notable classes in the GUI package are the beans used to dynamically construct the equation builder buttons and button groups. The equation(s)canvas, the classes that allow the equation and cursor drawing and se-
lecting, are respectively used in the equation builder and recursive specification frames. Lastly the CharacterSelector(Listener/event) class is used in the derive frame to be able to select a single token after selecting a term.

7.3.2 Util

The Util package is a collection of useful utilities. These utilities can be used everywhere. Most of these classes are used, structurally, in certain places and could not be made internal. As a consequence, there is no appreciable structure in the Util package.

The Util package contains:

- Exceptions, supporting the error handling.
- \LaTeX and file name handling, for OS independence.
- Data visualization tools for GUI (cell renderers for lists).
- Data containers used for inter-package communication.

The two Exception classes in the Util package are LatexException and PrimitiveTypeException. The LatexException was constructed to indicate a problem with the \LaTeX environment, which is needed as precondition to run PAT. PrimitiveTypeException indicates that an error has occurred during automatic parameter calculations.

LatexSymbol creates the \LaTeX images to be used on the GUI, by calling the external \LaTeX installation. NameHandler and DisAllowedCharacters deal with file creation in an OS independent way.

There are four ListCellRenderers for the display of e.g. icons in comboboxes. Each of these ListCellRenderers is specialized for a specific data type.

TermEdge is a class that encapsulates the notion of an edge in a DAG. It does not have a reference to the Term itself, but rather an reference to the parent FA and an integer that indicates to which argument the edge leads.

BracketPair is a class that provides a latex symbol for the combination of two corresponding brackets. BracketPair is the only TheoryEntity that is not part of the Algebra package (or its subpackages). It is not part of the package because it has no functionality apart from visualization.

The Node interface introduces some structure to a string of tokens, by binding brackets, functions and commas, and makes cycling through them possible.

7.4 Parser

In this section the parser’s internal process is described in detail, after which an example is given. The process of creating a DAG modulo associativity is also explained.

In sections 4.1.3(fig. 4.4) and 6.3.1 the two functions of the equation parser (List2TreeParser) were described. First, the equation parser checks whether an equation (represented by a list of tokens, a.k.a. string) is correct. Second, it returns the syntax tree. This is because it is assumed that if a syntax tree can be constructed, the equation is correct. Therefore, the equation is checked by construction. Note that construction here implies the use of all tokens in the
string. If leftovers remain, the construction failed. As described in section 6.3.1 these syntax trees are DAGs.

The parsing process can be described by the syntax accepting Labeled Transition System (LTS) shown in figure 7.6. There are several points to be noticed about this LTS. The black LTS elements are the parts that do the parsing. The grey elements facilitate the black parts. The dotted lines mark recursion. The black (and dotted) elements are explained first.

![Figure 7.6: Parser accepting automaton.](image

The first step in the parsing process is a case distinction on the class of the object: Bracket, Constant, Variable and Function. As the List2TreeParser is conceptually not a part of the rewriter, the communication is not restricted to the AbstractTypes interfaces. In this case distinction, for example, the AbstractTypes can not be used as they do not include the Bracket.

Brackets are always just accepted (skipped) except when they are the opening and closing tokens in the string. In that case, both brackets are removed, because these brackets do not add structure to the syntax tree. Constant and Variable tokens are always accepted. In the case the token encountered is a function, it is first checked whether the function is bracketed or an operator. For every argument in a bracketed function, the argument is removed from the string. Then, the parsing procedure is recursively called on the argument, and the result is added to the function object. After this the brackets are removed. In case the function is an operator, the next argument is processed as described above. Next, if the operator was binary, the trailing argument is processed. It is assumed that binary operators are always in infix-notation. Note that whether
or not the function was bracketed, the function token remains in the string.

After a token is accepted, and is in the accepting state marked with a check mark, the process continues with the next token, iterating over the string. To prevent ambiguous situations regarding different operator functions, priorities are introduced. Currently every function has a static priority. It is not strictly necessary that this priority is static. This may be changed in the future, and may depend on information in Function or FA. Note arguments need to be excluded from that decision, because it is this parsing process that binds arguments to their functions and one may not assume the output to be the input of the same process (Historical fallacy). To facilitate handling of these priorities, the process iterates over the input string multiple times, starting with the highest priority (0). The priority imposed on these Functions implies that a function is only parsed if it has the current priority level. Lower and higher priority functions are skipped and if the function was bracketed, the bracket is also skipped. Note that the recursive calls respect the current priority level.

In figure 7.7, the parsing process discussed is depicted from the point of view of the string. In this figure, it can be seen how a pointer iterates over the list of tokens. In the first iteration the pointer skips the + operator because the | operator has a higher priority. The right-hand side argument of | is bracketed, and is therefore processed recursively. Note that in figure 7.7 and all figures of DAGS below, the DAG representation abstracts from the FA reference to its function, and simply depicts the function in the FA.

**DAG modulo Associativity**  The DAG resulting from the process described above can, in combination with VariadicFA, be rewritten into a DAG modulo associativity (DAG/a). The main advantage of a DAG/a is that users do not need to apply the associativity axioms on terms. To this end a flag was introduced in Function “isAssociative”, which is set during Theory loading. This flag, in combination with the function name, is used to indicate which nodes in a DAG are merged to create a DAG/a. Depth first traversal is used; every node is either merged with the parent in case the same associative function is encountered, or recursively called to continue the process. This algorithm is visually depicted in figure 7.8.

When a white (undiscovered) node is discovered it is coloured grey. The node remains grey during investigation of the node and its children. After investigation the node is colored black. When two neighboring grey nodes are
merged their children do not change colour. DAG/a is currently not used, section 7.5.1 explains in detail why.

### 7.5 Rewriter

After the user obtains a syntactically correct equation, the user can decide to start a derivation. To enable this, PAT sets up a derivation environment using the constructs explained in section 7.2.1[Derivations] and an accompanying GUI. This environment calls the rewriter when all information is gathered. The results are displayed by the environment.

#### 7.5.1 Rewriting Process

The rewriting process in the Rewriter is called via the Apply method. The Rewriter class has little other functionality, most of which is error handling. The rewriting process is described according to its internal structure. This structure is depicted as a flowchart in figure 7.9. Method calls to (private) methods inside Rewriter are marked by parallelograms. Throughout the process there is a global boolean $B$ which is checked in the choices (rhombus marked by $B$?), and written by the methods (arrows marked by $B$!).

The central part of the rewriting process is unification. Unification is a well-known concept since Robinson published his algorithm [19] in 1965. Sev-
eral variants on this algorithm have been proposed. The Rewriter implements
the version described in “An Almost Linear Robinson Unification Algorithm”
[20]. This algorithm was chosen because it is a optimization of the Corbin and
Bidoit version [6]. Corbin and Bidoit showed that, although their version has
an exponential worst case, it has a better average time than the algorithms of
Paterson & Wegman [17] and Martelli & Montanari [15], with respective linear
and near-linear worst cases.

One of the main features of this version of Robinson’s unification algorithm
is that the algorithm is split up into two procedures; the unification, and the
post-occur check. The post-occur check checks that the resulting DAG does not
contain a cycle, and is only called if the unification succeeds. This structure is
retained and expanded as can been seen in figure 7.9.

Before the unify the Rewriter is initialized, the input is optionally dagified,
and the input is checked. Between the unify method and the post-occur check,
the Rewriter checks all the context conditions and substitutes according to the
unification. The term’s root needs to be set, as the complete term needs to be
checked in the post-occur check, not just the unified sub-term.

Figure 7.10 depicts the effects of the rewriting process phases, as shown in
figure 7.9, by applying axiom A6 to the term $X + 0$ in the term $(X + (X + 0))\|0$.
The initialization phase does not modify the DAGs; figure 7.10[a] shows the
selection of the edges that are to be rewritten, one in the equation and one of
the two roots of the axiom.

Dagify is a depth first tree traversal algorithm that turns a tree into a DAG.
Dagification is skipped in figure 7.10 because both trees are already in DAG
form. The boolean that decides to skip dagify is set during Rewriter object
creation.

In every unification step the types of the nodes under consideration are
checked. Depending on the type, and whether the axiom side node is formal,
it tries to bind, else the (syntactic) names are checked. A formal node is a
node that is variable on the axiom abstraction level of equational reasoning.
This means that they never may exist in an equation. Only formal nodes have
the ability to bind. Formality is not type bound, for example take $\tau$ and $a$.
which are of the type Action, yet the a-actions residing in axioms are formal,
while tau-actions are not. In figure 7.10[b] the first step of the unification is
Figure 7.10: Rewriting equation “(X + (X + 0))|0” with axiom A6 “Y = Y + 0”.

depicted: unification starts on the axiom root edge, in this case corresponding to the right-hand side of the axiom. Steps [c] and [d] continue the unification on the children; note that in step [c] variable “Y” is assumed to be “X”, therefore Y binds to X. When unification encounters a bound formal node that needs to bind with yet another node that is not equal to the already bound node, then rewriting fails. For example, when it is tried to unify y + z with x + x from axiom A3 (x + x = x) and the first x of the axiom is bound to y, the second x needs to bind with z, which fails because x is already bound to y.

Step [e] depicts the condition check phase. There are no conditions here, so this trivially succeeds.

The substitution, depicted in step [f], only changes the left root edge of the axiom from Y to X because Y is unified with X. Substitution starts on the other root edge as the one on which unification started, in this case the root edge corresponding to the axiom’s left-hand side. If substitution encounters a formal node that is not bound rewriting fails.

Step [g] shows the result after the selected equation edge is replaced by the axiom root edge opposing the selected axiom root edge.

The last step ([h]) depicts the result after the post occur check. The post occur check makes sure there are no loops in the resulting DAG. If loops are found rewriting failed. Note that the Post occur check is done on the complete DAG, not just on the rewritten part. The post occur check was not specified in “An Almost Linear Robinson Unification Algorithm” [20], but suggested a solution; the current implementation is based on [7, Paragraph 22.3].

Figure 7.11 shows another case of the rewriting process. In this case R.(p + q)|W.1 is rewritten using CM5. Note that CM5 has a condition if γ(a, b) = c,
and that $\gamma(R, W) = C$ is defined. Figure 7.11[a] depicts the starting state, and
[b] depicts the state after unification. Next is the condition check phase, which is
interesting in this case. Note that in the unification phase both formal actions $a$
and $b$ were bound to $R$ and $W$ respectively. Note also that $\gamma$ is a theory-universal
known auxiliary function. The condition check phase calls the “AuxCallCond”
object supplied with the Axiom’s Context. The condition retrieves actions $R$
and $W$ via the supplied $a$ and $b$. Then the condition binds the result of the
auxiliary function call of the actual $\gamma$ object to the supplied $c$ action, if the
result is non-null. This can be seen in [c] as the binding $c := C$ is depicted.
Step [c] depicts the situation after substitution. It shows the substitution of
$\{c := C; x := p + q; y = 1\}$. In [d] the result is shown after the root is set, and
the post-occur check is completed.

A side effect of the rewriting process as described above is that several axioms
and conditions can only be rewritten and evaluated in one direction. Again take
the condition $if \gamma(a, b) = c$ which is an equality, with a value on the one side,
and a function on the other side. Evaluating the function may or may not result
in a value equal to the stated value, which is decidable. But if one would try
to rewrite using the axiom’s other side, PAT would need to find a combination
of arguments that would yield the given value. This may result in several
correct resolutions. User interaction is needed to make this decision. This same
problem also occurs in axioms. For example, if an equation is rewritten with
axiom $0 \cdot x = 0$ (A7) from right to left, the formal variable $x$ is left unbounded.
Again user interaction is needed to bind the variable, because in this case any
term is valid. These one-way axioms are always rewritable in PAT from left to
right, but not from right to left. Other one-way axioms are CM1, CM4, CM6,
CS3, CS10, LM1, LM2, T4 and D3.
**Equivalence Check**  The equivalence check inspects both sides of the equation on syntactical equality, i.e., it checks isomorphy on the syntax of supplied terms. The equivalence check is not part of the rewriting process. It is however part of equational reasoning. Therefore, it is automatically done after every successful rewriting step, on both conclusions of the different equation sides. The check is done after every step, otherwise the user would need to initiate the check. This extra user interaction is unnecessary as it does not add to a better user experience.

**The problem with DAG modulo Associativity**  No elements of the rewriting process or equivalence check were designed or implemented for use with a DAG/a. This is due to a decision problem inherited from information that was abstracted away. This problem can easily be solved by user interaction. It is not clear whether the advantages of the reduced number of derivation steps due to the use of a DAG/a outweigh added complexity in the user interface and the rewriter. In which case the user needs to decide how to bind the formal variables.

The main problem with rewriting a term in DAG/a form and an axiom is that there are cases for which there is not just one solution. A typical case is depicted in figure 7.12. The figure depicts the unification of equation \( P + Q + R \) with a term \( m+n \) from an axiom, with both possible solutions \( b_1 = \{ m := p; n := q+r \} \) and \( b_2 = \{ m := p+q; n := r \} \).

![Figure 7.12: Two solutions (b1 & b2) of the unification modulo associativity problem (a).](image)

**7.5.2 de Bruijn**

The implementation of the process described in section 7.5.1 has no package imports besides the interfaces from the AbstractTypes package (section 7.1), and some java native data structures. Thus the containment described in section 6.3.3 is in effect.

In the rewriter, most methods are private and all attributes are private. The non-private methods are synchronized, to counter interleaving of method invocations. Thus the rewriter is a black box from the user’s (and PAT’s) point of view. This decreases the possible complexity that can arise from external calls.

All error messages are communicated via an indirect pipeline. The rewriting process pushes the error messages in the pipeline just before returning `null`. It is the responsibility of the using process to read out these messages and flush the pipeline after reading so it will not read the messages twice. This method
of error message passing does not break the de Bruijn criteria.

To conclude;

- The rewriter uses the minimal set of interfaces (the abstracts), as a minimal language of communicating the proof (the proof representation).

- The rewriter is an isolated proof kernel because:
  - It is a blackbox from the perspective of the user, with one method providing input and output, and an error channel.
  - The rewriter has no knowledge of the outside world.

- The rewriter is small; it has 750 lines of source code and comments. In contrast, Coq’s kernel has over 10,000 [10].

Therefore PAT’s rewriter conforms to the De Bruijn criterion.
Chapter 8

Future Work

In this chapter some points and fields are highlighted that could benefit from further investigation and / or future implementation. The subjects are grouped together in three categories:

- **PAT extensions**
  - Design and implement the algebra input (latex parser).
  - Design and implement the process that follows completion of a proof of an equation. Think about Data persistency (logging, printing, saving), use of the proof in other proofs, and program shutdown.
  - Design and implement the ability to use and create tactics (e.g. automatic linearization).
  - Implement the Recursive Specification (+validity check).
  - Implement standard communication.
  - Further design and implement non-axiomatized theorems like RSP.
  - Design and implement user interaction enabling term binding for one-way axioms. For example: Auxiliary Function Call Conditions are currently only checkable in one direction. Thus in a derivation where an axiom with such an condition needs to be used, there is a moment in the derivation process when the derivation can continue on only one side.

- **Parameters**
  - Implementing additional Data Algebras; Reals, Booleans and an abstract set “data”.
  - Check parameter names against all other parameter names of other types. In other words, make sure it can not happen that there are two variables “n”, one of type natural, and one of the type Process.
  - Make a parameter editor, which calls a equality check on value and type in the deriver.
  - Parameterize Constants and Variables. This functionality is not yet accessible.
Future Work

- **Genericity**
  - Generalize Functions to be Polymorphic (like actions in axioms), to enable to define algebraic function-properties without binding the function. E.g. $x\square(y\square z) = (x\square y)\square z$ if $\square \in \{+,\mid\}$
  - Automatically check via the axioms if a function has a property like associativity.
  - Design a Term-DAG modulo AC (associativity & commutativity), currently only modulo $A$ is present.
  - Design a Term rewriter that can handle Term-DAGs modulo $A$ and/or AC.
Bibliography


Appendix A

Manual

A.1 Installation Requirements

The following programs are required to be installed before PAT can be run.

- TU/e’s MikTeX Distribution or equivalent.  http://www.win.tue.nl/latex/
  This includes:
  - MikTeX.  http://miktex.org/
  - Mathtime, a \LaTeX\ package.  http://miktex.org/packages/mathtime
  - “MTSY.PFB”, a commercially available font.

If all these programs are installed: check whether the bin-directories of \texttt{DVIPS}, \texttt{\LaTeX} and \texttt{GSview} are in the Operating-System’s \texttt{$PATH$}-environment-variable.

A.2 Graphical Walkthrough

This section consists of an ordered collection of screenshots of PAT, each with a description.
Figure A.1: The Choose-Theory screen lets users select which theory (algebra) to work in.

Figure A.2: The Equation-Editor screen lets users build an equation.
Figure A.3: The Define-Function-Parameter screen lets users create a parameter to be bound to the function.

Figure A.4: The Define-Set screen lets users create and add or remove elements from the set.

Figure A.5: The Define-Auxiliary-Function screen lets the user select and add mapping elements to or remove mapping elements from the function.
Figure A.6: The Induction screen lets users choose a variable to apply structural induction on, resulting in the depicted amount of derivation cases.

Figure A.7: The Derive screen lets users derive a proof for the stated equation via equational reasoning.
Figure A.8: The Recursive-Specification screen lets users build a recursive specification.
Appendix B

RSE correction

Stated below are the table (B.1) and lemma (B.0.1) as depicted in the reviewed version of BBR09 [1]. In this table version axiom RSE6 is wrong. This proof derivation of the lemma was used to create the reverse engineered RSE6 version.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Axiom</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\mathit{BSP}_\perp + \mathit{RSE}) (A))</td>
<td>(\Box \phi \mathit{x} = x) RSE1</td>
</tr>
<tr>
<td>((\mathit{BSP}_\perp + \mathit{GC}) (A))</td>
<td>(\mathit{false} \mathit{x} = \perp) RSE2</td>
</tr>
<tr>
<td>(\phi \mathit{x} = \perp)</td>
<td>(\phi \mathit{\perp} = \perp) RSE3</td>
</tr>
<tr>
<td>((\phi \mathit{x} + y) = \phi \mathit{\perp} (x + y))</td>
<td>(\phi \mathit{x} (\psi \mathit{x}) = (\phi \land \psi) \mathit{x}) RSE4</td>
</tr>
<tr>
<td>(\phi \mathit{\rightarrow} (\psi \mathit{x}) = (\phi \supset \psi) \mathit{x})</td>
<td>(\phi \mathit{\rightarrow} (\phi \mathit{x}) = \mathit{x}) RSE5</td>
</tr>
<tr>
<td>(\phi \mathit{\rightarrow} (\psi \mathit{x}) = (\phi \supset \psi) \mathit{x})</td>
<td>(\phi \mathit{\rightarrow} (\phi \mathit{x}) = \mathit{x}) RSE6</td>
</tr>
<tr>
<td>(\phi \mathit{\rightarrow} (\psi \mathit{x}) = (\phi \supset \psi) \mathit{x})</td>
<td>(\phi \mathit{\rightarrow} (\phi \mathit{x}) = \mathit{x}) RSE7</td>
</tr>
</tbody>
</table>

Table B.1: The process theory \((\mathit{BSP}_\perp + \mathit{RSE}) (A)\) (with \(\phi, \psi \in \mathit{FB}\)).

**Lemma B.0.1** \((\mathit{BSP}_\perp + \mathit{RSE}) (A) \vdash \phi \mathit{\rightarrow} \perp = \neg \phi \mathit{x} 0.\)

\((\mathit{BSP}_\perp + \mathit{RSE}) (A) \vdash \phi \mathit{\rightarrow} \perp = \phi \mathit{\rightarrow} (\mathit{false} \mathit{x}) = (\phi \supset \mathit{false}) \mathit{x} (\mathit{false} \mathit{\rightarrow} x) = \neg \phi \mathit{x} 0.\)

Table B.2 depicts the correct axioms from the original publication [2], note that RSE6 is named RSE7 here.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Axiom</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\phi \mathit{x}) \cdot y = \phi \mathit{x} (x \cdot y))</td>
<td>(\mathit{false} \mathit{x} = \perp) RSE1</td>
</tr>
<tr>
<td>((\phi \mathit{x}) + y = \phi \mathit{x} (x + y))</td>
<td>(\phi \mathit{\perp} = \perp) RSE2</td>
</tr>
<tr>
<td>(\phi \mathit{\rightarrow} \psi \mathit{x}) = (\phi \land \psi) \mathit{x})</td>
<td>(\phi \mathit{\rightarrow} (\phi \mathit{x}) = (\phi \supset \psi) \mathit{x}) RSE3</td>
</tr>
<tr>
<td>(\phi \mathit{\rightarrow} \psi \mathit{x}) = (\phi \land \psi) \mathit{x})</td>
<td>(\phi \mathit{\rightarrow} (\phi \mathit{x}) = (\phi \supset \psi) \mathit{x}) RSE4</td>
</tr>
<tr>
<td>(\mathit{true} \mathit{x} = x)</td>
<td>(\phi \mathit{\rightarrow} (\phi \mathit{x}) = (\phi \supset \psi) \mathit{x}) RSE5</td>
</tr>
<tr>
<td>(\phi \mathit{\rightarrow} \psi \mathit{x}) = (\phi \land \psi) \mathit{x})</td>
<td>(\phi \mathit{\rightarrow} (\phi \mathit{x}) = (\phi \supset \psi) \mathit{x}) RSE6</td>
</tr>
<tr>
<td>(\phi \mathit{\rightarrow} \psi \mathit{x}) = (\phi \land \psi) \mathit{x})</td>
<td>(\phi \mathit{\rightarrow} (\phi \mathit{x}) = (\phi \supset \psi) \mathit{x}) RSE7</td>
</tr>
<tr>
<td>(\mathit{true} \mathit{x} = x)</td>
<td>(\phi \mathit{\rightarrow} (\phi \mathit{x}) = (\phi \supset \psi) \mathit{x}) RSE8</td>
</tr>
</tbody>
</table>

Table B.2: Root signal emission.
Appendix C

AbstractTypes API

C.1 Package AbstractTypes

C.1.1 ActionType

ActionType is a special subclass of the FunctionType, and therefore indirectly extends the Type interface. Because FunctionType can be returned by the TermFunction interface, the objects implementing the ActionType interface can also be returned.

ActionType extends the BindableEntity interface, because all action labels are potentially bindable to another action label.

All extensions of Type are part of the typing system for the rewriter class. An implementing algebra should implement the types where applicable.

DETECTION

<table>
<thead>
<tr>
<th>public interface ActionType</th>
</tr>
</thead>
<tbody>
<tr>
<td>implements FunctionType, BindableEntity</td>
</tr>
</tbody>
</table>

C.1.2 ActualParameter

Interface to all objects that might be used as a parameter. All Objects that are intended to be used as a parameter should implement this interface.

This interface is used by the prover to retrieve encapsulated parts of information of the parameters.
**Declaration**

```java
public interface ActualParameter
```

**Methods**

- `getLatex`
  ```java
  public String getLatex() 
  ```
  - **Usage**
    * Getter of this parameter's latex representation. This is used to correctly display parametrized Terms after substituting new parameters.
  - **Returns** - String representing this parameter's latex representation.

- `getName`
  ```java
  public String getName() 
  ```
  - **Usage**
    * Getter of this parameter's name. This is used by the prover to differentiate between elements of the same type. All objects that have the same name are assumed to be syntactically equivalent and are bound in the same object and, if evaluated, result in the same value (or process).
  - **Returns** - String representing this unique parameter.

- `getParameterType`
  ```java
  public String getParameterType() 
  ```
  - **Usage**
    * Getter of this object's type. **Note:** the implementation should enforce that every instance of this class should always return the same string representation! These strings should be comparably equal up to the `equal()` method of the `String` class. In other words, if the String class’ equal method can *not* differenticate the strings representing the types then we assume them to be equal!
  - **Returns** - String representing the type of the implementing class.

**C.1.3 BindableEntity**

The class `BindableEntity` is an interface to represent variables, parameters, primitive expressions and primitive variables. Occurrences of `BindableEntity`s in equations must be matched to occurrences in axioms and vice versa. The methods defined here are basically setters and getters.
To accomplish the matching, some methods are needed and some administration is needed. It is necessary to keep track of TheoryEntitys and PrimitiveEntitys that are formal, and thus can bind. It is also necessary to keep track of the formal entities that are bound already and to which other entities these are bound.

**Declaration**

```java
class BindableEntity
```

**Methods**

- **getCorrespondingEntity**
  ```java
  public <E> E getCorrespondingEntity()
  ```
  - **Usage**
    * Returns the Corresponding Entity if this object is bound, in all other cases returns null. Typing information should be added when this method is implemented / overridden.
    - **Returns** - The Corresponding Entity as type E.

- **isBound**
  ```java
  public boolean isBound()
  ```
  - **Usage**
    * Check the property isBound.
    - **Returns** - True if this entity is Formal and bound, false if it is either actual or (formal and unbound).

- **isFormalEntity**
  ```java
  public boolean isFormalEntity()
  ```
  - **Usage**
    * Check the property isFormalEntity.
    - **Returns** - True if this entity is formal, false if it is actual.

- **release**
  ```java
  public void release()
  ```
  - **Usage**
    * Releases the binding if there was one.

- **setCorrespondingEntity**
  ```java
  public boolean setCorrespondingEntity(java.lang.Object pBTE)
  ```
  - **Usage**
    * This method binds this formal entity to a Corresponding Entity.
AbstractTypes – Sort

- Parameters
  * pBTE - This Object is the entity to bind to, therefore it should be of a type which is bindable to this object’s type.
  
- Returns
  - Returns a boolean that indicates whether the binding succeeded. The implementation may fail to bind. These cases should however be kept to a bare minimum by the implementor. One may for example return false if null was given as argument.

- Exceptions
  * java.lang.IllegalArgumentException - Thrown when the given argument is not bindable to this object.

- setIsFormalEntity
  public void setIsFormalEntity( boolean pIsFormal )

- Usage
  * Set isFormalEntity to true or false.

- Parameters
  * pIsFormal - boolean indicating whether this entity is formal or actual.

C.1.4 ConstantType

ConstantType identifies the Type as a constant. It extends the Type interface.

All extensions of Type are part of the typing system for the rewriter class. An implementing algebra should implement these types where applicable.

Declaration

```java
public interface ConstantType
  implements Type
```

C.1.5 FunctionType

FunctionType is a Type and therefore extends that interface. FunctionType is returned by the TermFunction interface when the getFunction method is invoked.

All extensions of Type are part of the typing system for the rewriter class. An implementing algebra should implement these types where applicable.
C.1 Package AbstractTypes

Declaration

```java
public interface FunctionType
    implements Type
```

C.1.6 Sort

Sort is the type of all types. Both Type and PrimitiveType extend Sort. It is currently the root of the type-tree.

Declaration

```java
public interface Sort
```

C.1.7 Term

The Term Interface is the interface used by the deriver. All classes which need to be treated as a process by the deriver need to implement this interface. Every Term can potentially have parameters.

All implementations of methods declared here must:

- Use a minimum of variables declared out of the method scope.
- May only call methods declared in this interface or it’s subinterfaces.

This requirement also holds in all the child interfaces of the Term interface.

This interface ensures that the deriver can get all the information it needs to derive correctly. The implementation restrictions enforce encapsulation, with the purpose of minimizing the code used by the deriver to meet the "de Bruijn"-criterium.

The getType method must return a class that extends Type representing the type of the implementing Term. Type and its sub interfaces are all empty, their sole purpose is to give type information. The implementing class should implement one of these empty interfaces and then have getType return this. WARNING: implementing several Types may result in undefined behaviour!
AbstractTypes– TermFunction

Declaration

```java
public interface Term
```

Methods

- **getNParams**
  ```java
  public int getNParams()
  ```
  - **Usage**
    - Returns the number of parameters of this term.
  - **Returns** - int number of parameters.

- **getParameters**
  ```java
  public ActualParameter getParameters()
  ```
  - **Usage**
    - Returns the array of parameters of this term. All parameters should implement the `ActualParameter` interface.
  - **Returns** - `ActualParameter` array containing this term’s parameters. In case no parameters are present, an empty array of length 0 is returned.
  - **See Also**
    - `AbstractTypes.ActualParameter` (in C.1.2, page 83)

- **getRootName**
  ```java
  public String getRootName()
  ```
  - **Usage**
    - Returns the name given to this term on instantiation. The root name is used as a unique identifier per instance; different names imply non-equality. This is at the level of mathematical objects treated by the deriver, not at the level of java instances.
  - **Returns** - String name

- **getType**
  ```java
  public Type getType()
  ```
  - **Usage**
    - Returns the (super) type of the encapsulated term.
  - **Returns** - String uniquely defined per class. Type is used as a unique identifier per class; different names imply non-equality. This is both at the level of mathematical classes treated by the deriver and at the level of java classes.
C.1 Package AbstractTypes

C.1.8 TermFunction

The TermFunction interface is an extension of the Term interface. It adds the functionality of retrieving the arguments from a function in a term. All arguments of a function must also be terms.

All classes that implement this interface should also implement the (empty) FunctionType interface OR own a reference to such an object.

Declaration

```java
public interface TermFunction
  implements Term
```

Methods

- `getArguments`
  ```java
  public List getArguments()
  ```
  - **Usage**
    - This getter returns the array of terms given as arguments to this (term)function.
  - **Returns** - Term array containing this function’s arguments. In case no arguments are present `getArguments` returns an empty array of length 0.

- `getFunction`
  ```java
  public FunctionType getFunction()
  ```
  - **Usage**
    - Gets the actual function object used by the class that implements this interface. The function object is used by the rewriter to bind polymorphic functions like the actions.
  - **Returns** - Returns an object of type FunctionType (or one of its sub interfaces, such as ActionType).

- `setFunction`
  ```java
  public boolean setFunction(AbstractTypes.FunctionType pFunction)
  ```
  - **Usage**
    - Method to set the Function in a FunctionApplication
  - **Parameters**
    - * pFunction - Function to set
  - **Returns** - Boolean indicating whether the function was correctly set. True is returned on successful execution, false otherwise.
C.1.9  **Type**

Type is the type of all algebra-types. FunctionType, ConstantType, ActionType and VariableType all inherit from this interface. All future additions of algebra types should also extend this interface.

**Declaration**

```java
public interface Type
    implements Sort
```

C.1.10  **VariableType**

VariableType is a sub type of Type. VariableType is also a sub type of the BindableEntity interface, because all variables can in principle bind to a Term.

All extensions of Type are part of the typing system for the rewriter class. An implementing algebra should implement these types when applicable.

**Declaration**

```java
public interface VariableType
    implements Type, BindableEntity
```
C.2 Package AbstractTypes.Constructs

C.2.1 Context

All elements which can belong to the context of a proposition need to implement this interface. Two known examples of classes implementing this interface are `Condition` and `Assumption`.

**Declaration**

```java
public interface Context
```

**Methods**

- `check`
  ```java
  public boolean check()
  ```
  - **Usage**
    * Checks whether this context (property) holds. Assumptions always hold, and conditions may or may not hold. A class that always returns false implies a contradiction.
  - **Returns** - Returns true if this context holds, otherwise false.

- `use`
  ```java
  public Term use(AbstractTypes.Term[] args)
  ```
  - **Usage**
    * Use the context. The implementation decides what needs to be done. This method is neither used nor invoked at the moment, designed for future use.
  - **Parameters**
    * `args` - An array of Terms, possibly empty.
  - **Returns** - Returns an array of Terms, possibly empty.

C.2.2 Proposition

This interface defines the common methods of all propositions. In this context, a proposition is a statement constructed of two separate terms, one on the left-hand side and one on the right-hand side, separated by the Equal provided by the theory.

This interface purposely does not give any type information. It is encouraged to make a
The PropositionTerm interface extends the Proposition interface. Overriden methods add type information to the methods in Proposition. The interface also provides typed set methods. The type added here is Term.

**Declaration**

```java
public interface Proposition
```

**Methods**

- **getContext**
  ```java
  public Context getContext()
  ```
  - **Usage**
    - Gets the context of this proposition.
  - **Returns** - Returns an array of objects implementing the context interface.

- **getLefthandSide**
  ```java
  public Object getLefthandSide()
  ```
  - **Usage**
    - Gets the left-hand side object of this proposition.
  - **Returns** - The left-hand side Object.

- **getRighthandSide**
  ```java
  public Object getRighthandSide()
  ```
  - **Usage**
    - Gets the right-hand side object of this proposition.
  - **Returns** - The right hand side Object.
Methods

- **getLefthandSide**
  public Term getLefthandSide() 
  
  - **Usage**
    * Gets the left-hand side Term of this proposition.
  
  - **Returns** - The left-hand side Term.

- **getRighthandSide**
  public Term getRighthandSide() 
  
  - **Usage**
    * Gets the right-hand side Term of this proposition.
  
  - **Returns** - The right-hand side Term.

- **setLefthandSide**
  public void setLefthandSide( AbstractTypes.Term lhs ) 
  
  - **Usage**
    * Sets the left-hand side of this proposition, only type Term is accepted.
  
  - **Parameters**
    * lhs - Term representing the left-hand side.

- **setRighthandSide**
  public void setRighthandSide( AbstractTypes.Term rhs ) 
  
  - **Usage**
    * Sets the right-hand side of this proposition, only type Term is accepted.
  
  - **Parameters**
    * rhs - Term representing the right-hand side.
C.3 Package AbstractTypes.Data

C.3.1 PrimitiveConstant

PrimitiveConstant identifies the Type as a primitive constant. It extends the PrimitiveType interface.

All extensions of PrimitiveType are part of the typing system for the rewriter class. An implementing algebra should implement these types where applicable.

Declaration

```java
public interface PrimitiveConstant
implements PrimitiveType
```

C.3.2 PrimitiveFunction

PrimitiveFunction is a PrimitiveType and therefore extends that interface. All extensions of PrimitiveType are part of the typing system for the rewriter class. An implementing algebra should implement these types where applicable.

Declaration

```java
public interface PrimitiveFunction
implements PrimitiveType
```

C.3.3 PrimitiveOperatorApplicationTerm

The interface PrimitiveOperatorApplicationTerm extends the PrimitiveTerm interface for the primitive operators with a function to retrieve the arguments of a function.

Declaration

```java
public interface PrimitiveOperatorApplicationTerm
implements PrimitiveTerm
```
C.3 Package AbstractTypes.Data

Methods

- **getArguments**
  ```java
generic PrimitiveTerm getArguments( )
  ```
  - **Usage**
    * This getter returns the array of terms that were given as arguments to this (term)-function.
  - **Returns** - Term array containing this function’s arguments. In case no arguments are present this method returns an empty array of length 0.

C.3.4 PRIMITIVETERM

The interface **PrimitiveTerm** fulfills the same role as the Term interface in the process algebras.

Declaration

```java
public interface PrimitiveTerm
```

Methods

- **getName**
  ```java
generic String getName( )
  ```
  - **Usage**
    * Returns the name given to this primitive term on instantiation. Name is used as a unique identifier per instance; different names imply non-equality. This is at the level of mathematical objects treated by the deriver, not at the level of java instances.
  - **Returns** - String name

- **getNumericPart**
  ```java
generic PrimitiveTerm getNumericPart( )
  ```
  - **Returns** - Returns the numeric part of this PrimitiveTerm
  - **Exceptions**
    * Util.PrimitiveTypeException -

- **getSort**
  ```java
generic PrimitiveType getSort( )
  ```
  - **Usage**
    * Gets the type of the algebra that this PrimitiveTerm belongs to.
  - **Returns** - The type of the algebra this term belongs to.
• **getType**
  ```java
  public <T extends PrimitiveType> T getType()
  ```
  - **Usage**
    * Returns the (super) type of the encapsulated primitive term.
  - **Returns** - String is uniquely defined per class. Type is used as a unique identifier per class; different names imply non-equality, both at the level of mathematical classes treated by the deriver and at the level of java classes.

• **getVariablePart**
  ```java
  public PrimitiveTerm getVariablePart()
  ```
  - **Returns** - Returns the variable part of this PrimitiveTerm

• **isEqual**
  ```java
  public boolean isEqual(AbstractTypes.Data.PrimitiveTerm otherTerm)
  ```
  - **Usage**
    * Determine if two terms are equal
  - **Parameters**
    * `otherTerm` - Term to compare with
  - **Returns** - `this == otherTerm`
  - **Exceptions**
    * Util.PrimitiveTypeException - is thrown when the solution could not be resolved. This exception is due to the use of reflection by the implementation.

• **simplify**
  ```java
  public void simplify()
  ```
  - **Usage**
    * Simplify the primitive expression contained in this `PrimitiveTerm`
  - **Exceptions**
    * Util.PrimitiveTypeException -

### C.3.5 PRIMITIVETYPE

**PrimitiveType** is the type of all primitive algebras. **PrimitiveFunction**, **PrimitiveConstant**, and **PrimitiveVariable** all inherit from this interface. All future additions of algebras the the primitives should also implement this interface.

**Declaration**

```java
public interface PrimitiveType
    implements AbstractTypes.Sort
```
C.3.6 **PRIMITIVE VARIABLE**

*PrimitiveVariable* identifies the *PrimitiveType* as a primitive constant. It extends the *PrimitiveType* interface. *VariableType* is also a sub type of *BindableEntity* interface, because all variables can in principle bind to a *PrimitiveTerm*.

All extensions of *PrimitiveType* are part of the typing system for the rewriter class. An implementing algebra should *implement* these types where applicable.

**Declaration**

```java
public interface PrimitiveVariable implements PrimitiveType, AbstractTypesBindableEntity
```

C.3.7 **PRIMITIVEWRAPPER**

*PrimitiveWrapper* should only be used to indicate an object encapsulating a *PrimitiveTerm*. This encapsulated *PrimitiveTerm* can then be obtained via the *PrimitiveOperatorApplicationTerm*’s *getArguments* method.

**Declaration**

```java
public interface PrimitiveWrapper implements PrimitiveFunction
```