Heinrich Heine
Universität
Düsseldorf

INSTITUT FÜR INFORMATIK
Softwaretechnik und Programmiersprachen
Universitätsstr. 1 D–40225 Düsseldorf

A Translation of B Predicates to Answer Set Programming

Masterarbeit

im Studiengang Informatik
zur Erlangung des akademischen Grades

Master of Science (M.Sc.)

vorgelegt von
Alexandros Chrisovalantis Efremidis

Beginn der Arbeit:	13. Oktober 2020
Abgabe der Arbeit:	27. April 2021
Gutachter:	Prof. Dr. Michael Leuschel
	Prof. Dr. Gunnar W. Klau

Selbstständigkeitserklärung

Hiermit versichere ich die vorliegende Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, den 27. April 2021

Alexandros Chrisovalantis Efremidis

Abstract

In this thesis I develop an additional constraint solving backend for the ProB tool, which translates B predicates to s(CASP), a form of Answer Set Programming. Furthermore, the presented framework implements an interface enabling B predicates to be solved by the s(CASP) engine within ProB.

This work particularly focuses on

- the implementation and overall design of the framework and
- on evaluating its performance by comparing it to the native, Kodkod and Z3 backend of ProB.

The implemented framework is capable of translating numerous B predicates to s(CASP) and is able to find solutions in cases where Kodkod and Z3 are unable to follow. However, the work's empirical evaluation shows that this new approach is not quite on par with the other employed backends in terms of performance. This is mainly due to the suboptimal implementation of predicates computing functions in s(CASP). Nevertheless, in some cases the s(CASP) engine showed promising performances.

This work poses a foundation for future development regarding the translation of B predicates to Answer Set Programming and s(CASP) specifically. Further, this framework can be used to aid the verification of other solvers' correctness.

Contents

1 Introduction

In today's world software is prominently involved in everyday life. As a reason of that one wants to ensure that computer systems related to a specification operate as intended. In that regard, it is crucial for some applications to be verified in order to avoid severe damage when being deployed. A program's verification is achievable by using formal methods. Formal methods enable one to proof via tool support some specified system's correctness in a mathematical way, and thus validating its desired behavior.

In this thesis I aim towards extending the existing base of constraint solvers for the **B-Method** [Abr96], used in the **ProB** [LB03, LB08] tool, by providing an additional constraint solving backend. In particular, this work implements a **Prolog** [CR93] framework translating B predicates to **Answer Set Programming** [LW92, Lif99, Lif19] to then be solved by **s(CASP)** [ACS+18] within ProB.

This thesis is structured as follows. Section 1.1 gives an overview of Prolog and Section 1.2 presents the concept of formal methods. Related work is discussed in Section 1.3. Furthermore, in Section 1.4 I introduce Answer Set Programming and the chosen implementation for this translator s(CASP). The main contribution of this work consists of Section 2 illustrating the translation process and Section 3 empirically evaluating this new approach and comparing it to already employed backends of ProB. Afterwards, future work is presented in Section 4 whereas Section 5 eventually concludes the presented work.

1.1 Prolog

The translator presented in this thesis is implemented in **Prolog** [CR93], which is a logic programming language. Prolog has been implemented in many dialects such as **Ciao** [BCC+97], **SICStus** [CWA+88], **SWI** [WSTL12] and others. This framework specifically utilizes both Ciao and SICStus Prolog. The following documentation regarding Prolog itself applies to either one of the aforementioned dialects. Foremost, Section 1.1.1 gives a brief overview of Prolog's general syntax. Afterwards, the theoretical approach of searching for models in Prolog is introduced in Section 1.1.2, which also constitutes the baseline for s(CASP).

1.1.1 Syntax

In Prolog there is just a single data type called **term**. A term can express several subtypes, which are shown in Figure 1.

Subtype	Example
Atom	hitchhiker
Number	42
Variable	Var
Compound term	hitchhiker(42, Var)

Figure 1: Subtypes of Prolog Terms.

Atoms are ground and immutable values. In general, atoms are merely words beginning with a lowercase letter, for instance `hitchhiker`. However, various operators such as equality, the logical disjunction or character strings wrapped in single quotes, e.g. `'hello there'`, are atoms too. **Numbers** express the primitive types float and integer. **Variables** are immutable in Prolog. Usually they begin with an uppercase letter. Though, a variable can be declared as *anonymous* by adding an underscore to its designation before the first character. The sole underscore is the anonymous variable which is always a singleton even if it is used multiple times in the same scope. **Compound terms** consist of a *functor*, which is an atom, and an arbitrary amount of terms as arguments. Therefore, an atom is also a compound term with arity 0, for instance `hitchhiker/0`. Compound terms containing no variables are called *ground*.

Furthermore, lists are a frequently used in Prolog. A list is oftentimes denoted by `[Head|Tail]`. The empty list can be expressed by `[]`.

A Prolog program consists of **rules** and **facts**. Rules or clauses of the form `Head :- Body` can be viewed as a logic implication `Body` $\Rightarrow$ `Head`, i.e. if the body is satisfied then the head is also true. Facts are essentially rules with an empty body, meaning that the body is true. For example, the first row of Figure 2 is a fact and rows two and three express a rule. Predicate calls are also called **goals**. Goals in some body can be connected together with conjunctions or disjunctions denoted by `,/2` and `;/2`.

```
1: is_list([]).
2: is_list([_|T]) :-
3:  is_list(T).
```

Figure 2: An Example for a Prolog Predicate.

Finally, a **predicate** as depicted in Figure 2 can be viewed as a collection of rules and facts which all share the same functor and arity.

1.1.2 Semantics and Solving

Consider the following straightforward Prolog program portrayed in Figure 3.

```
p :- a, b.
a :- b.
b.
```

Figure 3: A propositional logic Prolog program.

As mentioned above, a Prolog program consists of Prolog predicates. From a semantic perspective, most Prolog predicates can be viewed as first-order logic predicates. Hence, the Prolog clause `p :- a, b.` corresponds to the disjunction of literals

$$a \wedge b \Rightarrow p \equiv p \vee \neg a \vee \neg b \tag{1}$$

in a pure logical sense. By virtue of just one literal being positive, the clause is called a **Horn** clause [Hor51]. A Prolog program can hence be seen as a theory of horn clauses. Inference rules can be applied to reach further conclusions on the base of a theory's premises.

In general, **inference rules** take an arbitrary number of premises into account yielding a conclusion. For example, *modus ponens* is an inference rule from propositional logic, which takes two premises of the form $p \Rightarrow q$, p and results in q. Another rule of inference utilized in propositional as well as in predicate logic is **resolution** [DP60]. Resolution allows one to check whether a logic formula is satisfiable or not by refutation. That is to say in a similar but more general way to modus ponens, resolution combines two clauses and produces a new clause called *resolvent*. The resolvent contains all the literals of the two original clauses except for complementary ones, as depicted in Figure 4. In case of being unable to eliminate complementary literals, resolution fails.

Clause 1	**Clause 2**	**Resolvent**
p	$\neg p$	*success*
$\neg p$	$\neg p$	*failure*
$\neg p \vee q$	p	q
$p \vee q$	$a \vee b \vee \neg q$	$p \vee a \vee b$

Figure 4: An Example for Resolution.

In the first example of Figure 4 the first clause corresponds to the theory p and the second one to the *denial* $\neg p$, expressing 'does p hold?'. The resolution of those clauses begets success, hence concluding that p indeed holds since the two complementary literals are eliminated. Failure is shown in the second example, as no literal can be eliminated when applying resolution. By looking at the third example one can see that modus ponens is, as a matter of fact, a special case of resolution.

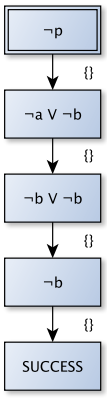
In logic programming the elemental rule of inference that is employed is **SLD** resolution [Kow74, AVE82], a refinement of plain resolution. The acronym stands for **S**election rule driven **L**inear resolution for **D**efinite clauses indicating that this approach uses a selection function for literal selection

and operates on definite clauses, i.e. Horn clauses, in a linear manner. In the instance of Prolog, the leftmost literal is always selected first, while backtracking is employed in case of failure. Furthermore, linearity is generally comprehended as 'derive a new denial and forget the old one'. An algorithm [ZS69, Lov70, Luc70] for linear resolution for some program or theory of Horn clauses in conjunctive normal form (CNF) is given in Figure 5.

1. Select some denial δ
2. Perform resolution on a clause 'query' of the theory and δ obtaining the resolvent δ'
   - If $\delta' = \textit{success}$ then terminate
   - Else set $\delta = \delta'$ (forget the old denial) and continue with step 2

Figure 5: An Algorithm for Linear Resolution [ZS69, Lov70, Luc70].

Anew consider the Prolog program in Figure 3. A query in Prolog consists of some goal which also may be a compound of multiple goals. In order to illustrate the aforementioned concept, the Prolog query `?- p.`, i.e. the denial $\neg p$ or the question whether p is satisfiable given the knowledge base, yields the SLD tree shown in Figure 6. Albeit Prolog actually uses an extension of regular SLD resolution the following example still applies.




```
p :- a, b.
a :- b.
b.
```

Figure 6: The SLD Tree for the Prolog Program of Figure 3.

An SLD tree portrays the process of SLD resolution for some program, where its root marks the initial denial. As the resolution of $\neg b$ and b at the bottom node results in success, p holds. The empty sets located at the tree's arcs are the respective most general unifiers, which is discussed subsequently.

Prolog programs in practice do not consist of propositional logic predicates only. Hence, first-order

logic programs as shown in Figure 7 must also be supported by SLD resolution.

```
p(A) :- a(A), b(A).
a(0).
a(1).
b(1).
```

Figure 7: A predicate logic style Prolog program.

Before SLD resolution is applicable to the program of Figure 7 substitution and unification has to be introduced. A **substitution** is a mapping of variables to terms. More precisely, some substitution of the form $\{A_1/t_1, \ldots, A_n/t_n\}$ suggests that variable A_i is replaced by term $t_i, i \in \{1, \ldots, n\}$. Furthermore, $t_a\sigma = t_b$ denotes the application of some substitution σ to term t_a yielding the substituted term t_b. A **unifier** of two terms t_c, t_d is a substitution σ so that $t_c\sigma = t_d\sigma$ holds. Examples for substitutions and unifiers are given in Figure 8.

Term 1	**Term 2**	**Possible Unifier**
$p(A)$	$p(a)$	$\{A/a\}$
$p(X, Y)$	$p(q(a), B)$	$\{X/q(a), Y/B\}$
$p(a)$	$p(b)$	*not unifiable*

Figure 8: Examples for Substitutions and Unifiers.

A **composition** $\sigma_1\sigma_2$, with $\sigma_1 = \{A_1/a_1, \ldots, A_n/a_n\}$ and $\sigma_2 = \{B_1/b_1, \ldots, B_n/b_m\}$, of two substitutions is defined as:

$$\sigma_1\sigma_2 = \{A_1/a_1\sigma_2\} \cup \{B_i/b_i | B_i \notin \{A_1 \ldots A_n\}\} \tag{2}$$

An example for a composition is $\{A/B, C/c\}\{B/b, D/d\} = \{A/b, B/b, C/c, D/d\}$. Moreover, a substitution σ_1 is *more general* than σ_2 if there is a third substitution ρ and $\sigma_1\rho = \sigma_2$. For instance, $\sigma_1 = \{A/a\}$ is more general than $\sigma_2 = \{A/a, B/b\}$ due to $\rho = \{B/b\}$ and $\sigma_1 = \{A/B\}$ is more general than $\sigma_2 = \{A/a, B/a\}$ since there exists $\rho = \{B/a\}$.

In practice, one is interested in the **most general unifier**, *mgu* for short. Some substitution φ is called an mgu for terms t_1, t_2 if φ is a unifier and there is no other unifier that is more general than φ. Robinson showed that if two terms are unifiable there always exists a unique mgu, which can be computed in linear time [Rob65]. Further algorithms have been proposed throughout the years [PW76, MM82]. However, this thesis does not focus on the peculiarities of unification. Therefore, the foregoing algorithms are not discussed in detail.

By making use of unification one is able to perform resolution on terms and clauses in predicate logic. An algorithm and an illustration is provided in Figure 9 [Rob65].

Algorithm

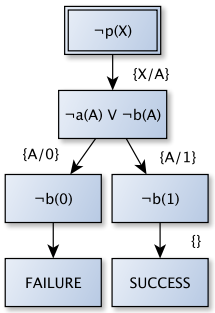
1. Select query $p(a_1, \ldots, a_n)$ and denial $\neg p(b_1, \ldots, b_n)$ of clauses C_a, C_b
2. In order to avoid clashes rename all variables of clause C_a
3. Compute mgu θ of the query and denial
4. Apply θ to clauses C_a, C_b
5. Execute resolution of propositional logic on $C_a\theta$ and $C_b\theta$ and set the resolvent to $(C_a\theta \cup C_b\theta) \backslash \{p(a_1, \ldots, a_n)\theta, \neg p(b_1, \ldots, b_n)\theta\}$

Example

1. $C_a = p(A) \vee q(A), C_b = \neg p(b)$ and select query $p(A)$ and denial $\neg p(b)$
2. No clashes can occur in this case
3. mgu $\theta = \{A/b\}$
4. $C_a\theta = p(b) \vee q(b), C_b\theta = \neg p(b)$
5. Resolvent equals $q(b)$

Figure 9: Resolution on Clauses of Predicates Logic using Unification [Rob65].

Recalling the Prolog program of Figure 7, one can now perform SLD resolution on predicate logic programs. The SLD tree of Figure 10 shows the aforesaid mechanism with `?- p(X).` as the initial query. As one can see, failure is encountered in the left leaf, as the resolution of denial $\neg b(0)$ fails. Hence, the algorithm backtracks to the previous choice point continuing with the next possible sub-query $\neg b(1)$, which eventually leads to proving that the initial denial $\neg p(X)$ holds.




```
p(A) :- a(A), b(A).
a(0).
a(1).
b(1).
```

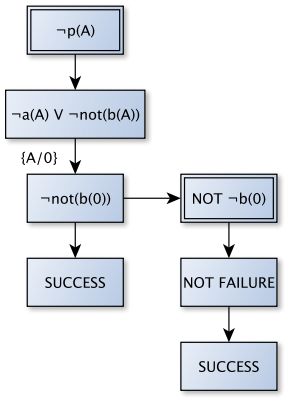
Figure 10: The SLD Tree for the Prolog Program of Figure 7.

SLD resolution allows for deriving conclusions from a theory or program of Horn clauses in propositional as well as in predicate logic. Nevertheless, the restriction to Horn clauses restrains one to use negated calls or literals in the body of a rule. Consequently, Prolog uses an enhanced version of habitual SLD resolution called SLDNF resolution. Although the two approaches are sometimes referred to as the same one they differ in reality.

SLDNF resolution enables one to incorporate negation, more precisely **Negation as Failure** (NF)

literals, in a rule's body additional to SLD semantics [CL89, AD94]. Negation as Failure is a *non-monotonic* [MD80] inference rule which amounts to declaring a negated ground literal `not(p)`, or in case of Prolog `\+p`, as false if the engine is unable to prove that `p` is true given the knowledge base [Cla78]. When a negated literal, `not(p)` for instance, is encountered in the SLDNF inference process, the solver deploys a new SLDNF computation with `not(p)` as the initial denial returning true if the subproof `p` fails or false in case of success. In some cases `not(p)` might be semantically unequal to the logical negation $\neg p$ with respect to NF, as SLDNF resolution is incomplete given an arbitrary Prolog program [Llo12]. Furthermore, in case of NF deriving failure for a negated goal of the form `not(p(`$t_1, \ldots, t_n$`))` no bindings for variables located in $t_1, \ldots, t_n$ are returned.

The concept of NF is closely related to the **closed-world assumption** stating that a truthful statement is *known* to be true, whereas *unknown* knowledge is deemed to be false [Rei81]. On the other hand, there is also the converse **open-world assumption**, which suggests that some statement irrespective of knowing its truthfulness may still be true [DS06]. In perspective of actual programming, an open-world assumption system may return *unknown* for some query indicating a lack of knowledge for which a system using the close-world assumption would return *false* disregarding its dependence to the possibly limited amount of knowledge.

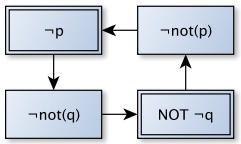



```
p(A) :- a(A), \+b(A).
a(0).
a(1).
b(1).
```

Figure 11: An Example for a SLD(NF) Tree.

By slightly altering the program of Figure 7 one can demonstrate the way SLDNF resolution operates. The SLD(NF) tree of Figure 11 shows how SLDNF resolution obtains the result for a negated goal by emitting a sub-branch in the derivation.

As a general remark, SLD(NF) resolution per se does not come with any precautions regarding maintaining consistency with respect to infinite loops or non-termination. Consider Figure 12's SLD(NF) cycle, since in this case the depicted graph is not a tree anymore, where it is shown that for the query `?- p.` SLDNF resolution is unable to derive a terminating solution. The graph shows the first derivation cycle. However, the emitted sub-branch invokes at its core $\neg p$ yielding an infinite loop.




```
p :- \+q.
q :- \+p.
```

Figure 12: An Example for *non-termination* for SLD(NF) resolution.

Finally, Prolog implementations are inter alia known for providing an interface to **Constraint Logic Programming** (CLP). CLP enables for reasoning with and about constraints in logic programming [JL87]. SICStus Prolog, for instance, includes the CLP(FD) [COC97] library. CLP(FD) allows for defining constraints over finite domain integers. In particular, integer variables are bound to an interval which is propagated throughout the program considering all other reciprocal variable relations. Thus, the initial domain gets continuously restricted to a final interval from which a legal assignment is available satisfying the underlying constraint or otherwise resulting in a contradiction. Moreover, the major use cases of constraint logic reasoning over integers or numerical values in general are declarative arithmetics and solving combinatorial problems. Arithmetical CLP operations, for instance, are generally designated by adding # as a prefix to ordinary operators. Furthermore, ideas of CLP regarding rationals and reals have been suggested by Holzbaur [Hol95]. Figure 13 shows CLP's fundamental advantage over standard arithmetic reasoning. The program on the left hand side throws an *instantiation error*, whereas the one on the right hand side succeeds with correct bindings for all variables.

```
p :-
 A = 1,
 C is A + B,
 B = 2.
```

Instantiation Error

```
p :-
 A = 1,
 C #= A + B,
 B = 2.
```

Succeeding

Figure 13: An Example for CLP(FD) in Prolog.

1.2 The B-Method and ProB

This section is structured as follows. Formal methods are introduced in Section 1.2.1. Section 1.2.2 highlights the B-method, which is incorporated in the ProB tool. ProB, a model checker and disprover for the B-method, is eventually discussed in Section 1.2.3. The layout of this introduction is partially inspired by Dunkelau [Dun17].

1.2.1 Formal Methods

In practical and industrial use cases software testing is prominently involved in order to verify software's correctness. However, it may prove difficult debugging code in a mature state when not taking any precautions regarding validating correctness during implementation. Therefore, a way to counter this issue is making use of a specification or requirements describing desired structural and behavioral properties for some piece of software before beginning with the actual implementation phase. This can be achieved by employing formal methods.

Formal methods are mathematically based techniques describing system specifications in the field of software and requirements engineering [Win90, WLBF09, Gho00]. They aim towards designing *zero defect* software typically by providing the means of precise requirements definition regarding implementation, consistency, completeness and correctness [Mil93]. The relation between client and implementor is often called *design by contract* [Mey92], i.e. a software's realization must satisfy its specification.

In general, a desired specification within the realm of formal methods is expressed by a formal specification language. A **formal specification language** provides mathematical notions for meticulous articulation and representation with respect to specifying software's behavior. Hence, the pairing of implementation and specification serves as a correctness theorem. Examples for languages are ASM [Bör03] or the B-method [Abr96].

A formal method solely cannot prove a software's correctness. Hence, they primarily come with *tool support* for validating formulae expressed by formal specification languages. Thereby, formal methods are able to actively aid the development process.

In practice, formal methods and their application address a broad verity of pragmatic considerations [Win90]. During the development phase formal methods provide means for revealing ambiguities, inconsistencies and incompleteness. Therefore, if used in an early stage, they disclose design flaws without even necessarily running the underlying application that otherwise may be discovered during testing yielding a costly production. Furthermore, if used later, the intended behavior of the underlying software is thereby confirmable for a stable release.

1.2.2 The B-Method

The **B-Method** is a formal method and contains **B**, a formal specification language, for development and verification of software systems [Abr96]. B offers expressive capabilities for software reasoning and is based on the following three subjects [CM03, Hoa05]:

1. Sets are used for data modeling, which are expressed by the Zermelo-Fraenkel set theory with the axiom of choice [FBHL73],

2. state modifications are characterized by general substitutions [Abr96], which allows for deterministic and nondeterministic transition and
3. models at different levels of abstraction are denoted by the refinement calculus [BW12].

Two instances for tool support are Atelier B [Ste] and ProB [LB03, LB08]. The latter is further discussed in Section 1.2.3.

Apart from classical B, the B-method offers additionally Event-B, which serves as an extension to the traditional formalism [Abr10]. However, in the scope of this work discussing classical B suffices. Therefore, the peculiarities of Event-B are not further considered.

Furthermore, the B-Method provides the means for expressing a system in a theoretical way, called abstract machines. **Abstract machines** are generally used to describe specifications or implementations by providing a space of state and means for transitions (operations).

As a general remark, since this thesis is primarily concerned with translating B predicates to Answer Set Programming the focus lays more towards expressing predicates than machines. Therefore, their layout and syntax are merely covered briefly.

Figure 14 shows the basic layout of an abstract machine and provides a simplistic example. An abstract machine comprises of clauses specifying the machine's behavior. An arbitrary machine is structured as follows. Note that the presented structure is incomplete, however sufficient for introductory purposes. Consider 'The B-book' by Abrial for further reading [Abr96]. **MACHINE** contains the abstract machine's designation. The clause **SETS** expresses set definitions, thus constructing custom data structures. **CONSTANTS** and **VARIABLES** are lists of identifiers, which are respectively set in **PROPERTIES** and **INITIALISATION**. **INVARIANT** constraints a machine's state. Formally, an invariant I is a conjunction of the form $I = i_1 \wedge \cdots \wedge i_n$ with i_j denoting predicates. An invariant is intended to be satisfied in every reachable state. Finally, **OPERATIONS** defines all possible transitions available. Altering the state when executing a transition may lead to a violation of the invariant, as variables can possibly be assigned to invalid values. Hence, operations constitute proof obligations, which need to be satisfied either automatically or interactively supported by the underlying proof engine [CM03]. In perspective of a machine's consistency, one is concerned with the *establishment* of its invariant. That is to say, for an invariant I applying the general substitution S, emitted by some operation, the invariant preservation predicate states that the invariant must still hold, thus expressed formally by

$$I \Rightarrow [S]I \tag{3}$$

declaring S establishes I.

Considering the depicted example in Figure 14 one is able to get a first glance of how abstract machines are defined in practice. This particular machine specifies a simplistic instance of an electric car. The only variable in its realm is the battery charge status, which is initially set to 5. The car possesses just two operations enabling it to move and recharge a battery unit respectively at

```
MACHINE
 header

SETS
 sets

CONSTANTS
 identifiers

PROPERTIES
 predicate

VARIABLES
 identifiers

INVARIANT
 predicate

INITIALISATION
 substitution

OPERATIONS
 operations
```

Layout

```
MACHINE
 electric_car

VARIABLES
 battery

INVARIANT
 battery : NATURAL

INITIALISATION
 battery := 5

OPERATIONS
 move =
  BEGIN
   battery := battery - 1
  END;

 recharge =
  PRE
    battery < 5
   THEN
    battery := battery + 1
   END
 END
```

Example

Figure 14: An Example for basic B-Machines.

a time. When executing a transition with the operation *move* the *battery* variable gets substituted by its original value decremented by one. It is evident that at some point after repeatedly executing *move* the invariant is violated due to *battery* leaving the space of natural numbers. In order to counter this imminent violation, the car features a way to recharge its battery with the operation *recharge*. However, in contrast to the previous operation this one contains a precondition denoted by **PRE**. Essentially, it states that some condition has to be satisfied for state to be modified.

1.2.3 ProB

ProB is an automated analysis toolkit for the B-method enabling for animation, model checking and constraint solving [LB03, LB08]. ProB allows for unveiling possible errors of the underlying specification. The tool is used in a broad field of industrial applications, most notably in railway control. Since its launch in 2003 ProB has been developed further to a large extent. In order to clarify some of ProB's most important features, model checking is briefly introduced.

Model checking is an automated technique used to verify program properties and thereby find violations for a model's specification by traversing the reachable space of states [CJGK+18, CES86, QS82]. Usually, model checking is applied in an exhaustive manner. In particular, the entire reachable state space outgoing from an initial state is visited by transitioning consecutively to the next available one(s). The concern of consistency may vary depending on the use case. For instance, one could show interest in checking wether specific properties, such as an invariant, hold for the underlying model or in looking for dead-locks.

ProB is capable of deploying two major approaches for automated consistency checking:

1. **Model checking** allows for invariant violation detection, which is utilized exhaustively in ProB. However, in order for this approach to succeed sets and integer variables need to be restricted to small domains or cardinalities [LB03]. Thereby, it is feasible to indeed traverse every possible state of the model's respective space of states. Nonetheless, this feature also provides the means for sophisticated debugging and testing with a non-exhaustive execution. The user is able to set upper bounds for state traversing.

2. **Constraint-based checking** on the other hand, instead of sequentially visiting every possible state this approach analyzes each transitional operation on its own validating whether it may cause an invariant to be violated irrespective of the initial state. Inspired by the ALLOY analyzer [Jac02, JSS01], the engine interprets the invariant preservation predicate as a constraint over all possible substitutions applicable. That is to say, by using the constraint-based checker as a disprover one desires to refute the constraint for a fix invariant I

$$\neg(\exists S : I \Rightarrow [S]I) \equiv \forall S : I \wedge \neg[S]I \tag{4}$$

thus showing whether I can be established by at least one general substitution S.

Furthermore, ProB supports refinement checking [LB05], LTL model checking [Pnu77] and bounded model checking [BCC+03]. However, in the scope of this thesis I do not dig deeper into these topics. As pointed out in Section 1.2.2, I further focus on B predicates met by the constraint-based checker rather on the model checker.

1.3 Related Work

This section presents related work, which essentially consists of the variety of the constraint solving backends offered by ProB.

The core of the ProB kernel is implemented in SICStus Prolog [CWA+88] with the incorporation of co-routines and CLP(FD) [COC97] as its interface to CLP. Vanilla ProB seems to thrive when solving enumerable problems [KBL14]. This is perhaps due to CLP(FD) featuring the means for providing enumeration over variables in form of integer intervals. However, in some cases the native ProB backend poses a suboptimal option regarding time consumption and decidability. For instance, when encountering a predicate containing variables restraining one to use enumeration, native ProB might struggle to find a solution [KBL14]. An example for this certain instance is dealing with variables bound to infinite domains. Consider the predicate

$$a, b \in \mathbb{N} \land a < b \land b < a \tag{5}$$

which is obviously false. In this case, native ProB is unable to find the contradiction [DKS19]. Since the nature of native ProB is applying enumeration to the underlying problem, the solver assigns the interval $[1, \infty)$ to the variables a, b. Therefore, the engine tries to compute every value pair assignment determined to find the one satisfying the constraint.

Consequently, ProB provides additional constraint solving backends to remedy those instances where the native backend might encounter difficulties. The two major engines Kodkod and Z3 are presented. Both of them come with their respective strengths, weaknesses, special properties and deficiencies.

1. **Kodkod** is a constraint solver for first-order logic based on SAT and supports relations, transitive closure and partial models [TJ07, Tor09]. However, KodKod within ProB does not support operations on sequences and higher-order types, such as sets of sets. Depending on the problem or even the way of expressing it, Kodkod is able to outperform ProB's native backend noticeably, for instance when relational operators and universal quantification are used to find solutions for sets [PL12]. Contrarily, native ProB operates much faster for arithmetical problem instances which is one of Kodkod's weakest points [PL12]. Furthermore, due to Kodkod's underlying SAT nature it is unable to handle infinite domains in contrast to ProB.

2. **Z3** is an SMT solver employing the DPLL(T) algorithm [DMB08, GHN+04]. Similar to Kodkod, some B predicates and expressions cannot be translated efficiently to SMT-LIB [RT06,

BST+10], and are thus not supported by the integrated translation of ProB. Empirical evaluation shows that neither of the two solvers outperforms the other, rather there exist numerous instances where just one of them is capable of deciding whether the given formula is satisfiable [KL16]. In order to illustrate the aforementioned insight, ProB performs better on higher-order operations, such as set comprehensions, whereas Z3 decides with ease that the predicate of Equation (5) is not satisfiable [KL16].

1.4 Answer Set Programming

This section is devoted to the introduction of Answer Set Programming. In Section 1.4.1 I give an overview of Answer Set Programming in general. Further, I point out some of its implementations and dialects in Section 1.4.2. Lastly, Section 1.4.3 presents s(CASP), the chosen dialect for this thesis.

1.4.1 What is Answer Set Programming?

As the title of this section suggests, this segment is prominently inspired by the work 'What is Answer Set Programming?' by Lifschitz, wherein an exceedingly perspicuous introduction to Answer Set Programming is given [Lif19].

> '**Answer Set Programming** (ASP) is a form of declarative programming oriented towards, primarily NP-hard, search problems.'–[Lif19]

Generally, ASP is concerned with finding *stable models* also referred to as *answer sets* for the underlying problem. This approach is originally based on the application of non-monotonic reasoning incorporating ideas of default logic [Rei80] and autoepistemic logic [Moo84] to negation as failure.

An ASP program P is a finite set of clauses, where each rule $r \in P$ is of the form

$$r = h \leftarrow t_1 \wedge \cdots \wedge t_m \wedge neg\ t_{m+1} \wedge \cdots \wedge neg\ t_n \tag{6}$$

with the head h and the body's literals $t_1, \ldots, t_n$ being compound terms [Lif19, Fit92, BET11].

Most ASP dialects share syntactical similarities with Prolog and therefore, at first glance, look somewhat alike. Nevertheless, their semantics noticeably diverge. This design of logic programming is motivated by the incompleteness of SLDNF resolution regarding negation as failure [Llo12]. In order to remedy that restraint, one desires the introduction of classic or also called *strong* negation. In particular, the keyword *neg* may express NF as well as strong negation in ASP.

```
1: cross :- not train.
2: cross :- -train.
```

Figure 15: An Example for NF vs. strong negation [Lif19].

Assuming one wants to cross a railway, it is naturally a case of imperative importance that a train is not crossing that certain railway at the same time. Figure 15 demonstrates the way strong negation may be useful in practice where the first row is representing NF and the second row represents strong negation. When asserting the first row, it is stated that one is clear for crossing while being in disregard of actually knowing whether a train passes the railway. In other words, if the program lacks the information whether a train is passing, then NF yields the same conclusion as knowing that the tracks are clear. However, the second row represents the aforementioned assumption in a more accurate way. The second row states that when indeed knowing about the train's absence the railway is safe to cross. Hence, NF and strong negation behave differently in case of insufficient knowledge. Consequently, ASP systems enable one to express knowledge in the close-world assumption as well as simultaneously leaving some other predicates in the realm of the open-world assumption, in contrast to Prolog.

Furthermore, an interesting property of all kinds of ASP systems is that the process of finding answer sets, unlike SLDNF resolution utilized by Prolog, in principle always terminates [Lif19]. In most of ASP's implementations this feature is due to the employment of an enhanced version of the DPLL algorithm for finding solutions, hence stable models [DP60, DLL62]. These approaches are somewhat similar to some SAT solvers as proposed in the work of Gomes et al. [GKSS08]. The DPLL algorithm is essentially a complete procedure (SAT solver) for proving or refuting a propositional logic formula's satisfiability in CNF. It is safe with respect to non-termination, i.e. it terminates for n Boolean values in $\mathcal{O}(2^n)$. However, some dialects employ other methodologies while maintaining the overall intended ASP conduct.

As previously stated, ASP solvers endeavor to find stable models as the solution for some ASP program. Let F be some propositional formula and M be a set of literals, which appear in F. Now truth is assigned to the literals in M and false to the rest of F's literals. The so-called *reduct* F^M is the formula obtained by replacing each maximal sub-formula not satisfied by M with falsity [Fer05]. If the reduct F^M is satisfied, then M is a model of F. Furthermore, a model M is called *stable* if there exists no proper subset of M satisfying F^M. It is straightforward that every stable model is also a model relative to some formula F. Also, if F is not satisfied by M, then $F^M = \bot$ holds. In essence, a **stable model** is a minimal model satisfying a propositional logic formula [Fit92].

The tabular of Figure 16 highlights the differences between models and stable models for propositional formulae in CNF. The third example depicts the set $M = \{p, q, s\}$, which is a model of $F = p \wedge (q \Rightarrow (s \wedge \neg t))$. However, since there exists a proper subset $\{p, s\}$ satisfying F, it is not a stable model.

In order to find stable models in practice for an ASP program P one needs to agree to the fol-

Formula F	Truth value set M	Reduct F^M	Classification
$p \wedge q$	$\{p\}$	$p \wedge \bot = \bot$	*no* model
$p \wedge q$	$\{p, q\}$	$p \wedge q = \top$	*stable* model
$p \wedge (q \Rightarrow (s \wedge \neg t))$	$\{p, q, s\}$	$p \wedge (q \Rightarrow (s \wedge \neg \bot)) \equiv p \wedge s = \top$	*stable* model
$p \wedge (q \Rightarrow (s \wedge \neg t))$	$\{p, s\}$	$p \wedge (\bot \Rightarrow (s \wedge \neg \bot)) \equiv p \wedge s = \top$	*stable* model

Figure 16: An Example for Stable Models.

```
domain(1).
domain(2).
inc1(A,B) :-
 domain(A),
 B is A + 1.
inc2(A,B) :-
 B is A + 1.
```

Figure 17: An Example for safe and unsafe rules.

lowing convention. Assuming a classical ASP implementation, thus deploying a form of the DPLL algorithm, one may view P's clauses as a collection of propositional logic formulae in CNF. However, analogous to Prolog ASP programs oftentimes do not come with propositional rules only. As a result, ASP systems typically introduce an initial grounding phase replacing P by all the ground instances of its clauses with respect to all variables' related domains yielding the substituted ground program P_G. However, the grounding phase succeeds only if every clause of P is safe. A rule is considered as safe in case every variable in its body occurs in some positive literal (compound term) thereby specifying the variable's finite domain. A rule is deemed unsafe otherwise. For instance, `inc1/2` of Figure 17 is safe, whereas `inc2/2` is unsafe due to not specifying `A`'s, and thereby `B`'s, domain. Finally, the computation of a stable model for the grounded program P_G may be attempted.

1.4.2 Implementations of Answer Set Programming

In the course of recent years various dialects of the ASP paradigm have been proposed. This section briefly presents some of ASP's available implementations and highlights their peculiarities and weak spots.

Classical implementations, as pointed out in Section 1.4, usually involve a grounding front-end producing a ground program, which is solvable by an ASP engine. Lparse [Syr00] is a popular grounder for ASP originally designed for Smodels [SN01] but is used by numerous ensuing solvers, such as ASSAT [LZ04], Cmodels [LM04], gNt [JN04], nomore++ [AGL+05a, AGL+05b] and further implementations. A more recent design, extending Lparse by inter alia adding aggregation functions, is the grounding tool gringo [GST07]. The answer sets of gringo's output program are computed by the solver clasp [GKNS07]. Moreover, both mechanisms have been integrated into

one monolithic tool called clingo [GKK+08], thus making the entire process more user friendly.

However, traditional approaches come with several disadvantages. Smodels or clingo, for instance, do not support explicit data structures such as lists, which are used on a regular basis in logic programming. Although lists may be expressed by compound terms indirectly, their absence impairs the programmer's convenience. Another and certainly more impactful drawback is the general restraint of free variable usage. Since every rule needs to be safe, one is restricted to finite domains in the sense of every possible value has to be included in the grounded program. For problems with vast domains this may lead to the instance that no suitable ASP program is reasonably obtainable for the underlying program or to the case of a combinatorial explosion regarding the program's size [ACS+18]. To illustrate this issue one may consider the results of the fifth Answer Set Programming competition [CGMR16]. As an example, a 5GB ground file was generated for a 60 job incremental scheduling problem, whereas the size soared to 50GB when doubling the job amount. Given that industrial applications have oftentimes to deal with instances multiple orders of magnitude larger with a superlinear increase of memory than the example above, it is evident that a *ground-and-slove* approach is not always appropriate [FFS+18]. This concern is also called the *grounding bottleneck* [BLS13].

In order to tackle the aforementioned matter, one may employ lazy-grounding to avoid a possibly immense grounding phase. The idea behind lazy-grounding is to calculate a desired ground instance for further computation when they are needed during runtime instead of grounding the entire original program. For instance, Alpha [Wei17], ASPeRiX [LN09], GASP [DPDPR09] and Omiga [DTEF+12] incorporate this idea. Additionally, some of these more recent implementations, for example GASP, support some built-in data structures, such as lists, and outperform prior presented designs, lparse + smodels in particular, for some benchmarks [DPDPR09]. While the utilization of tools using lazy-grounding is beneficial regarding memory size they, nevertheless, underperform in terms of time consumption in most practical cases [FFS+18]. Hence, deciding which ASP strategy to deploy for some arbitrary problem could cause a dilemma.

However, in order to be more flexible with respect to problem declaration one may lean towards choosing a lazy-grounding paradigm with built-in explicit data structures. Furthermore, an additional extension to ASP incorporating CLP may be desirable to broaden its expressiveness. A recently implemented dialect of ASP introduces CLP to ASP, also called CASP [MGZ08], with no grounding discussed below in Section 1.4.3.

1.4.3 s(CASP)

s(CASP)[1] is a novel Answer Set Programming implementation coalescing stable model semantics and Constraint Logic Programming [ACS+18]. It is build upon the s(ASP) execution model, an ASP interpreter written in Prolog. s(CASP) inherits and generalizes s(ASP) while remaining parametric with respect to CLP.

[1] https://gitlab.software.imdea.org/ciao-lang/sCASP

s(ASP) is a recently developed query-driven implementation of Answer Set Programming [MSCG17]. Unlike ordinary ASP systems, s(ASP) does not employ any SAT based methodology and, due to its Prolog heritage, does not rely on a grounding phase either prior or during execution. Hence, s(ASP) admits the usage of variables without specifying their respective domains, whereas classical ASP approaches would consider the program as unsafe. s(ASP) deploys a top-down, query-driven procedure virtually an extended version of SLDNF resolution for evaluating programs under the ASP semantics. By virtue of s(ASP) being goal-directed, the engine just computes a partial stable model, which is the portion required for answering the underlying query. Partial stable models are also further referred to as stable models for simplicity. In essence, this approach can be thought of Prolog with complementary properties supporting ASP.

s(ASP) provides four relevant attributes deviating from Prolog including [MSCG17]:

1. The assumption of an **extended Herbrand universe**,
2. the computation of a **dual program**,
3. the involvement of **constructive negation** and
4. the incorporation of **coinduction**.

These properties deal with subsuming ASP semantics, i.e. negation and precautions for infinite loops.

s(ASP) assumes that variables of unsafe rules, hence variables without any specified domain, take values in an *extended* Herbrand universe. An **extended Herbrand universe** hereby implies that for a global constraint expressed by s(ASP)'s negated existential quantifier `:-`, equivalent to $\neg\exists$, the program demands knowledge regarding the quantified goal in order to be satisfied. **Global constraints** can be placed manually and are asserted by the engine in some cases. They essentially imitate and, thus ensure a correct behavior regarding negation in ASP [ACCG20]. A meta-interpreter executes the global constraints after the query. Figure 18 demonstrates that both shown programs are semantically equivalent with respect to first-order logic. The first program is comprehended in the same way a regular Prolog program is evaluated. Thus, the goal of the form `married(A)` succeeds for any `A`. However, in the second example the interpreter needs to have evidence that every single possible instance of `A` satisfies the call `married(A)`. Clearly, the amount of instances is infinite. Therefore, the second program possesses no stable model. Combining both statements yields a program, which is semantically equivalent to `married(A)` regarding satisfaction. Yet, the stable models diverge. For instance, the query `?- married(john).` resolved by the original program returns the stable model `{married(john)}` and alternatively `{married(john), married(Var)}` for the combined version.

Negation in s(ASP) is generally treated differently compared to Prolog, and to negation as failure in particular. s(ASP)'s interpreter resolves negated goals of the form `not p` against dual rules. **Dual rules** allow for non-ground negated calls, `not p(A)` for example, to return variable bindings

s(ASP) Program	Semantics
`married(A).`	$\forall A : married(A)$
`:- not married(A).`	$\neg\exists A : \neg married(A) \equiv \forall A : married(A)$

Figure 18: An Example for Differences between Prolog and s(ASP) under the extended Herbrand Universe semantics [ACS+18].

1. Let $\overrightarrow{x} = x_1 \ldots x_n$ be a tuple of n fresh variables for each predicate p/n.
2. For each i-th rule $p_i(\overrightarrow{t_i}) \leftarrow B_i$ with $i = 1 \ldots k$, let $\overrightarrow{y_i}$ be the tuple comprising of all variables occurring in B_i but not in $\overrightarrow{t_i}$ and alter p_i/n representing $\forall\overrightarrow{x} : (p_i(\overrightarrow{x}) \leftarrow \exists\overrightarrow{y_i} : B_i)$ by renaming the variables of $\overrightarrow{t_i}$ with $\overrightarrow{x}$.
3. Each predicate and its clauses is transformed by applying *Clark's completion* [Cla78] to:
   - $\forall\overrightarrow{x} : (p(\overrightarrow{x}) \leftrightarrow p_1(\overrightarrow{x}) \vee \cdots \vee p_k(\overrightarrow{x}))$
   - $\forall\overrightarrow{x} : (p_i(\overrightarrow{x}) \leftarrow \exists\overrightarrow{y_i} : b_{i1} \wedge \cdots \wedge b_{im} \wedge \neg b_{im+1} \wedge \cdots \wedge \neg b_{in})$.
4. By definition the predicates semantically equivalent *duals* are:
   - $\forall\overrightarrow{x} : (\neg p(\overrightarrow{x}) \leftrightarrow \neg(p_1(\overrightarrow{x}) \vee \cdots \vee p_k(\overrightarrow{x})))$
   - $\forall\overrightarrow{x} : (\neg p_i(\overrightarrow{x}) \leftarrow \neg(\exists\overrightarrow{y_i} : b_{i1} \wedge \cdots \wedge b_{im} \wedge \neg b_{im+1} \wedge \cdots \wedge \neg b_{in}))$.
5. Finally, the result is obtained by applying *De Morgan's laws*:
   - $\forall\overrightarrow{x} : (\neg p(\overrightarrow{x}) \leftrightarrow \neg p_1(\overrightarrow{x}) \wedge \cdots \wedge \neg p_k(\overrightarrow{x}))$
   - $\forall\overrightarrow{x} : (\neg p_i(\overrightarrow{x}) \leftarrow \forall\overrightarrow{y_i} : \neg b_{i1} \vee \cdots \vee \neg b_{im} \vee b_{im+1} \vee \cdots \vee b_{in})$.

Figure 19: The Algorithm for the Computation of Dual Programs [ACS+18].

regardless of the positive goal `p(A)` succeeding [APS04]. The idea is to thereby emulate the computation of stable models as in a conventional ASP system, where everything is ground from the beginning. An algorithm computes the dual program prior to executing the query. Hence, the later evaluated program is actually the union of the original and the dual program. Furthermore, the dual program is also not resolved under plain SLDNF semantics. Some coinductive additions to SLDNF resolution are introduced related to the treatment of loops, discussed afterwards.

As elaborated in Figure 19, synthesizing a dual program is accomplished by first computing Clarke's completion, which assumes that the set of predicates completely captures the entire space for atomic formulae to be satisfied [Cla78]. Afterwards a definition for each dual predicate $\neg p(\overrightarrow{x})$ is obtained by applying De Morgan's laws. Figure 20 illustrates the algorithms functionality. The built-in predicate `forall/2` handles the evaluation of the universal quantifier which checks for the call `forall(V, G)` whether the goal `G` is satisfiable for all possible values of `V`. However, `forall/2` cannot be invoked in source code by the programmer.

```
a(A) :- A \= 1.
```

Figure 21: An exemplary s(ASP) program for Constructive Negation.

```
p(A) :- a(A), not b(A,B).
a(0).
b(1,1).
```

```
not p(A) :- not p1(A).
not p1(A) :- forall(B, not p1(A,B)).
not p1(A,B) :- not a(A).
not p1(A,B) :- a(A), b(A,B).
not a(A) :- not a1(A).
not a1(A) :- A \= 0.
not b(A,B) :- not b1(A,B).
not b1(A,B) :- A \= 1.
not b1(A,B) :- A = 1, B \= 1.
```

Figure 20: A exemplary s(ASP) program (left) and its respective Dual Code.

Furthermore, s(ASP) extends the common unification algorithm with constructive negation. **Constructive negation** is denoted by `\=`, intuitively expressing that for `A \= b`, `A` may be unified with any term except for `b` [MSCG17, ACS+18]. Thereby, the interpreter keeps track of which values for each variable are not unifiable, and thus allowing (dual) rules with negated equality to succeed with correct bindings. Since traditional ASP programs are completely ground ab initio, it is straightforward that within the program's finite realm an answer, regardless of the involvement of negation, is obtainable. Hence, constructive negation is essentially a feature mimicking this behavioral property of conventional ASP approaches. However, this subject poses a more challenging task in environments where variable domains are first off unrestricted. s(ASP) is able to compute a set of terms, which cannot be unified with the respective variable. Nevertheless, the engine is not capable of providing a concrete binding in contrast to classical ASP due to the absence of explicit domains. For instance, the query `a(A)` returns the binding `A \= 1` for the program of Figure 21. Moreover, constructive negation comes with some restrictions. Variables can only be constrained against ground terms with constructive negation. The disequality constraint fails otherwise.

Top-down execution models may suffer from non-termination in practice. Techniques such as tabling have been proposed for prototype systems to enhance termination properties [MG12]. However, the presence of constructive negated calls introduces further challenges. In s(ASP) **non-monotonic checking rules** are latently deployed during the execution. Thus, the process maintains consistency concerning global constraints and infinite loops. s(ASP)'s non-monotonic checking rules are rooted in **coinduction**, more precisely in coinductive SLD resolution [SMBG06, SBMG07]. The following precautions are taken to deal with breaking infinite loops over negation. Three kinds of non-termination are considered, i.e. even, odd and positive loops. The implemented procedures are briefly exhibited subsequently.

Odd loops occur when the call graph contains a cycle with an odd number of intervening negated goals, where the encountered goal closing the cycle unifies with an ancestor call [ACS+18]. Odd

loops yield global constraints restricting the stable model. Hence, in order to avoid contradictions and to satisfy the induced global constraints, s(ASP) introduces non-monotonic checking rules. For every clause or global constraint of the form

$$\forall \overrightarrow{x} : (p_i(\overrightarrow{x}) \leftarrow \exists \overrightarrow{y_i} : B_i \wedge \neg p_i(\overrightarrow{x})) \tag{7}$$

either $\neg B_i$ or the query $p_i(\overrightarrow{x})$ using another rule must hold. Therefore, for each of the aforementioned clauses, a rule of the form

$$\forall \overrightarrow{x} : (chk_i(\overrightarrow{x}) \leftrightarrow \forall \overrightarrow{y_i} : (\neg B_i \vee p(\overrightarrow{x}))) \tag{8}$$

is synthesized. Figure 22 shows an example for a global constraint and clauses featuring an odd loop. This idea is realized by deploying a static analysis of the call graph, which evaluates the number of negated calls throughout recursion. However, during run time when a call unifies with its respective negated representation, the execution fails and backtracks since the call is contradictory.

```
:- not q(A).
p(A) :- q(A), not p(A).
```

```
nmr_check :- chk1, forall(A, chk2(A)).
chk1 :- forall(A, q(A)).
chk2(A) :- not q(A).
chk2(A) :- q(A), p(A).
```

Figure 22: An exemplary s(ASP) program without a Stable Model (left) and its corresponding non-monotonic Checking Rules.

Even loops are essentially the same kind of odd loops but with an even amount of intervening negated goals [ACS+18]. Recalling the Prolog program shown in Figure 12, it is evident that the execution does not terminate under SLDNF semantics, whereas the syntactically equivalent s(ASP) program of Figure 23 indeed terminates.

```
p :- not q.
q :- not p.
```

Figure 23: The s(ASP) program equivalent to Figure 12.

Positive loops arise when a call unifies with an ancestor similar to the previous two kinds. However, in this case there are no intervening negative calls along the call graph [ACS+18]. Consider the example shown in Figure 24, which for the query `?- nat(A).` returns an infinite amount of results. s(ASP) identifies the underlying construct as a positive loop, and thus fails. Nevertheless, this behavior compromises completeness and soundness. Therefore, s(CASP) somewhat restores the aforementioned properties by investigating whether the encountered goal and its respective ancestor call are equal. In that instance, s(CASP) returns the first applicable answer, hence unifying `A` with `0` in the example.

```
nat(0).
nat(A) :- nat(B), A is B+1.
```

Figure 24: An Example for Positive Loops in s(ASP) [ACS+18].

Finally, s(CASP) can be thought of an extension and even an improvement to s(ASP). s(CASP) inherits parts of s(ASP) and its aforementioned properties, yet offers new features and reimplements a considerable amount in Ciao Prolog [BCC+97]. Since Prolog and s(ASP) share a lot of characteristics regarding variable treatment, s(CASP) lets the Ciao engine take care of all operations Prolog is able to handle natively instead of leaving them to the s(CASP) interpreter. Furthermore, s(CASP) introduces CLP in the form of Holzbaur's CLP($\mathbb{Q}$), which admits declaring constraints over rationals and integers [Hol95]. CLP($\mathbb{Q}$) is further used internally for an improved version of the interpreter. Additionally, s(CASP)'s constructive negation solver is a reimplementation of s(ASP)'s version, called CLP($\neq$), to cope with CLP($\mathbb{Q}$) constraints. The authors emphasize that s(CASP) easily outperforms s(ASP) by providing some prevalent benchmarks [ACS+18].

1.5 Motivation and Goals

As stated at the beginning of this thesis, the fundamental motivation is to develop *zero-defect* software with the help of formal methods. For instance, this is achievable by specifying a system's desired behavior with B abstract machines to then be validated by a software verification tool such as ProB. In order to model check a machine and verify its correctness, predicates are evaluated along the way. For instance, predicates in the B-method express a machine's properties, invariants and its initialization. Therefore, constraint solvers such as native ProB, Kodkod and Z3 are employed to support the overall verification process. As they all come with their respective strengths and weaknesses it seems natural to explore further options.

Answer Set Programming is a paradigm of growing interest in the recent years. Due to its declarative nature ASP poses an attractive alternative to already well-established constraint solving oriented methodologies. However, for this work an unconventional approach is chosen compared to traditional ASP. s(CASP) is independent of an initial grounding phase, and hence offers remarkable freedom in terms of expressiveness regarding variable's domains similar to Prolog. Moreover, s(CASP) provides supplementary features, such as constructive negation, which may prove beneficial in constraint solving.

Consequently, the following two subjects constitute the main contribution of this work:

- In this thesis I develop a Prolog framework that translates B predicates to s(CASP). Besides, I implement an interface integrating the translator in ProB. Thus, it enables one to call the s(CASP) engine within ProB, and hence to harness s(CASP)'s capabilities.

- Furthermore, I aspire to investigate s(CASP)'s performance via benchmark evaluation covering numerous varieties of predicates. The performance measurements are compared to the aforementioned already employed backends of ProB. Thereby, the grasp of s(CASP)'s proficiencies is ameliorated.

However, formulating a complete translation of the entire B realm presumably goes beyond the scope of a single master's thesis. Therefore, this work excludes the subsequent criteria:

- Infinite domains are not translated to s(CASP). Particularly, expressions such as $a \in \mathbb{N}$ or $b \in \mathbb{Z}$ are not considered. The task of incorporating CLP($\mathbb{Q}$) constraints in s(CASP) to enumerate over possible instances for numerical domains comparable to ProB's CLP(FD) backend may be subject to future work. Nevertheless, statements as `a = 1` for example generally imply in computer systems that the variable a is associated with integer. In that regard, s(CASP) advantages related to unrestricted variable usage do not elapse.
- Lastly, the vast majority of B predicates is covered in the translation framework. Yet, some of them are not incorporated, such as set summation, set product, iteration, closure, projection, lambda abstraction and the support of sequences.

The presented framework and its evaluation are thoroughly discussed in Section 2 and Section 3 respectively.

2 Translating B Predicates to s(CASP)

In the following, I elaborate on the developed translation framework. Foremost, Section 2.1 outlines the framework as a whole and illustrates it with a graphical sketch. Section 2.2 presents some examples and, finally, Section 2.3 provides a detailed description of the translation process.

2.1 Design and Workflow

This framework[2] consists of two major parts:

1. The **translator** obtains a (typed) abstract syntax tree (AST) representing a B predicate emitted by ProB and generates semantically equivalent s(CASP) code. It is written in SICStus Prolog to ensure compatibility and is constructed as a loadable extension package of ProB
2. The **interface** establishes the connection between the translator and the s(CASP) engine. Since s(CASP) is written in Ciao Prolog, a C interface links the two Prolog implementations together, thus enabling for solving a B predicate by s(CASP) within ProB.

[2] https://gitlab.cs.uni-duesseldorf.de/efraimidis/b-to-asp

The overall workflow is illustrated in Figure 25. An AST is expected as input by the framework, hence ProB parses the predicate prior to translating. This initial process is depicted as a black box in the flowchart. Thereafter, the translator generates a s(CASP) file containing executable code. When the translation is completed, the interface calls the s(CASP) engine and obtains the result. Lastly, some post-processing is applied and the result is returned back to ProB.

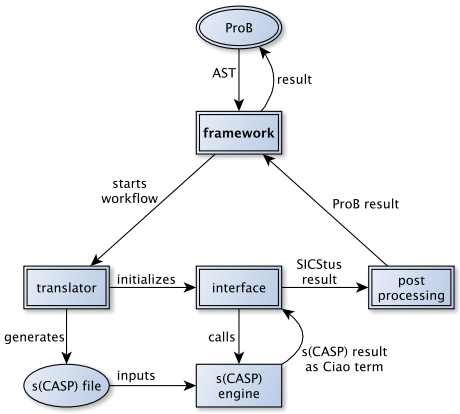


Figure 25: The framework's general workflow.

The translator gradually walks over the AST and computes for each encountered B node a semantically equivalent s(CASP) component. Through this process a s(CASP) program is recursively build. Examples in Section 2.2 portray the translation procedure in more depth.

Both Ciao[3] and SICStus[4] Prolog provide a native bi-directional C interface capable of converting Prolog terms to C structures and vice versa [Rit93]. Besides that, s(CASP) offers an interior interface allowing for calling the s(CASP) engine via Ciao Prolog. However, the crucial interface between the two Prolog dialects is missing. Hence, this work implements a Ciao/SICStus interface. By means of the native Ciao interface, the obtained result for the underlying predicate is translated to a C structure representing a Ciao term. The new developed fragment, linking Ciao and SICStus together, converts the Ciao representation to a custom C struct, which is used to generate an equivalent representation of a SICStus term in C. The native SICStus interface then transforms the acquired C term to a SICStus term in Prolog. Thus, the framework is able to process and pass the answer back to ProB.

2.2 Translation Example

In this subsection I illustrate how the framework operates for exemplary predicates.

[3] https://ciao-lang.org/ciao/build/doc/ciao.html/foreign_interface_doc.html
[4] https://sicstus.sics.se/sicstus/docs/3.7.1/html/sicstus_11.html#SEC128

Consider the B predicate of Equation (9).

$$a = 1 \wedge b = a + 2 \tag{9}$$

Figure 26 demonstrates the sequential translation procedure for the aforementioned predicate. More specifically, the translation tree shows the generated s(CASP) code for the B predicate in form of an AST.

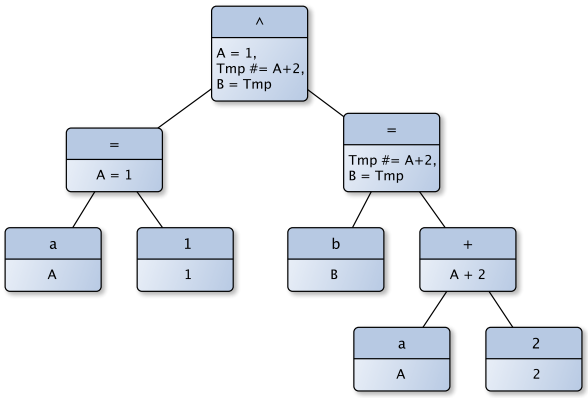


Figure 26: An Example for the Translation of a B predicate to s(CASP).

```
1: #include 'preliminaries.pl'.
2:
3: constraint(B, A) :-
4:  A = 1,
5:  Tmp #= A + 2,
6:  B = Tmp.
7:
8: ?- constraint(B, A).
```

Figure 27: The generated s(CASP) program for the Predicate $a = 1 \wedge b = a + 2$.

The actual generated s(CASP) code is shown in Figure 27. Due to s(CASP)'s query-driven nature, the predicate is wrapped in the body of the `constraint/2` predicate. Thus, the solver executes the query `?- constraint(B, A)`, indicated by row eight. The first row imports preliminary predicates, primarily containing s(CASP) representations of B predicates, which are further described in Section 2.3. Since s(CASP) comes with virtually no built-in predicates, the `preliminaries.pl` file contains inherent predicate implementations, such as `member/2` and `append/2`, which are generally used on a regular basis in logic programming. The s(CASP) engine returns the in Figure 28 shown answer for the program of Figure 27.

```
typedef struct {
 int type;
 char* term_atom;
 int term_int;
 float term_float;
 struct term_node* next_element;
 struct term_node* sub_list;
 int arity;
 struct term_node* next_argument;
 struct term_node* sub_structure;
} term_node;
```

Figure 29: The custom C Struct used to represent a generic Prolog Term.

```
QUERY:?- constraint(B,A).

        ANSWER: 1 (in 0.359 ms)

MODEL:
{ constraint(3,1) }

BINDINGS:
B = 3
A = 1 ?
```

Figure 28: The Answer for the s(CASP) program of Figure 27.

However, a printed answer is disadvantageous for further processing. The Ciao/s(CASP) interface returns a Prolog list containing the data of Figure 28. Amongst other things, a term is located in the list including the bindings. The partial stable model and further data except for the bindings are dispensable for the constraint solving task. For this particular instance, the term representing the bindings is of the form

```
[[B,A],[3,1]]
```
(10)

which requires further manual treatment to assemble an answer processable by ProB.

The two final steps before the answer is returned to ProB are firstly passing the bindings from Ciao to SICStus Prolog and afterwards employing some post-processing on the provided term. Since the interface's peculiarities regarding the overall converting process are inessential to describe in detail, merely the custom C struct representing a generic Prolog term is further discussed. As one can see in Figure 29, the struct features members to represent primitive data as well as lists and compound terms. Consider the interface's program files for more elaboration.

Depending on the underlying term to be translated, different values are set for a `term_node`. For example, an integer is denoted by assigning its value to the member `term_int` and setting `type = 1`,

which internally proclaims that the struct embodies an integer. The term of Equation (10) is transformed to the C struct illustrated in Figure 30. The unused members for each individual struct are omitted for simplicity.

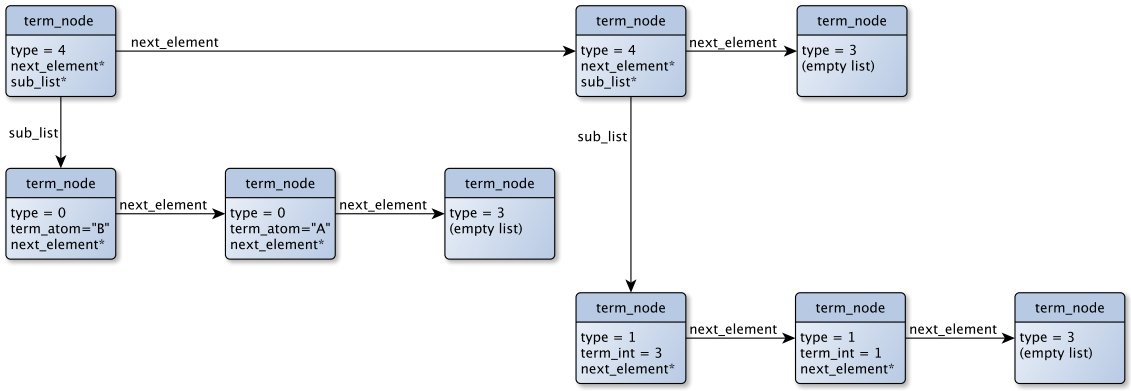


Figure 30: The illustrated custom C struct for the Bindings `[[B,A],[3,1]]`.

Afterwards, the aspired bindings are returned to the framework via the interface as a SICStus term. Finally, some post-processing is performed, which mainly serves the purpose of producing a syntactically suitable answer for ProB. In this occasion, the final result is of the form:

```
solution([binding(a,int(1),1), binding(b,int(3),3)])
```
(11)

Furthermore, the framework's post-processing handles computing solutions out of constructive negation terms. This feature excels the inherent capabilities of common logic programming languages limited to negation as failure, such as Prolog.

Consider the predicate

$$a \neq \{(2,4)\} \wedge a \neq \{(1,3),(2,5)\} \tag{12}$$

for which the technical details regarding translation are not repeatedly outlined. The s(CASP) interpreter's output merely expresses the disequality constraints but neglects to calculate a binding for a, as can be seen in Figure 31.

```
QUERY:?- constraint(A).

        ANSWER: 1 (in 0.206 ms)

MODEL:
{ constraint(A | {A \= [t(2,4)],
   A \= [t(2,5),t(1,3)]}) }

BINDINGS:
A \= [t(2,4)], A \= [t(2,5),t(1,3)] ?
```

Figure 31: The s(CASP) Answer for the predicate of Equation (12).

Since ProB expects a concrete binding for variables, a solution for the given constructive negation constraint has to be found. Accordingly, the post-processing component executes an intuitive algorithm, which receives a conjunction of disequality terms and computes a candidate satisfying the underlying disequality constraint. However, this method works on data structures for which at least one variable is bound to an integer only. Supporting booleans and especially strings requires greater effort, which may be subject to future work. As stated in Section 1.5, the only supported infinite data structure is implicitly denoted integers, e.g. $a = 1$, implying that $a \in \mathbb{Z}$. Hence, by agreeing to the convention that for the predicate $a \neq 1$ any other integer is a valid assignment for a, the following algorithm depicted in Algorithm 1 is applicable to any data structure containing one integer at a minimum.

Algorithm 1 A pseudo algorithm for computing constructive negation bindings.

Require: a conjunction of disequality terms C
1: find a term $t \in C$ containing an integer
2: replace all integers in t with the same fresh variable v obtaining t'
3: find the minimum m of all integers appearing in every $c \in C$
4: set $v = m - 1$
5: **return** t'

Since B is strictly typed, the predicate $a \neq \{(2,4)\} \wedge a \neq \{(1,3),(2,5)\}$ indicates that a's type is a set of integer tuples, whereas in pure logic it remains uncertain. Therefore, the aforementioned algorithm is able to use an arbitrary disequality term containing at least one integer as a template. For the predicate of Equation (12), the algorithm consequently computes the binding $a = \{(0,0)\}$.

2.3 Formal Description of the Translation

In this subsection I focus on the syntactical and semantical details of the translation. I cover the B predicates that are included in this translation in a gradual manner by generally following the order of a summary of the B toolkit by Robinson [Rob05]. For each of the supported predicates, I devise an equivalent s(CASP) representation. If necessary, I give a formal definition along with its

respective counterpart in B.

Throughout this section the following designations are used: P, Q denote predicates; E, F denote expressions; x, y denote single variables; z denotes a list of variables; S, D denote set expressions; U denotes a set of sets; m, n denote integer expressions; r denotes a relation and f a function. Furthermore, for any B code b, $\mathtt{T}_b$ denotes the translation of b in s(CASP). However, for the sake of simplicity the designations of B identifiers are also used for their representatives in s(CASP). For a B expression e the variable $\mathtt{V}_e$ denotes the desired result for code $\mathtt{T}_e$. The exact process of generating $\mathtt{T}_b$, $\mathtt{T}_e$ and obtaining $\mathtt{V}_e$ is further discussed subsequently.

2.3.1 Translation of Primitive Values

Foremost, I define in Figure 32 the way primitive values, i.e. an identifier or a ground boolean, number or string, originating from B are translated to s(CASP). However, these translation rules place some restrictions. Variables' designations in s(CASP) begin with an uppercase character. To simplify the translation process every character is converted to its uppercase representative, which somewhat restrains the freedom of variable names. Furthermore, variables beginning with `Tmp` denote internal temporary variables. However, one is not obliged to avoid using variables of the form `Tmp` as it is translated to `TMP` anyway.

Type	**B**	**s(CASP)**
Identifier	`id`	`ID`
Boolean	`TRUE`	`true`
Boolean	`FALSE`	`false`
Integer	`123`	`123`
String	`"string"`	`'string'`

Figure 32: The Translation of *Primitive Values*.

2.3.2 Translation of Predicates

1. **Conjunction**

B	**s(CASP)**
`P & Q`	$\mathtt{T}_P, \mathtt{T}_Q$

Figure 33: The Translation of *Conjunction*.

Although this translation depicted in Figure 33 seems lucid, one needs to consider a drawback of s(CASP). In pure logic the order of P and Q is irrelevant for the conjunction's semantics. However, the order matters in s(CASP) as $\mathtt{T}_P$ and $\mathtt{T}_Q$ have to be evaluated sequentially.

If a ground value is needed for evaluating T_P, which remains unbound until T_Q is evaluated, the evaluation of the code T_P, T_Q throws an instantiation error. Nevertheless, a considerable amount of cases where this issue occurs is dealt with s(CASP)'s CLP library.

2. **Disjunction**
   Let $x_1, \ldots, x_k$ be the variables occurring in `P` and `Q`.

B	s(CASP)
`P or Q`	`subconst(`$x_1, \ldots, x_k$`)`

Figure 34: The Translation of *Disjunction*.

For both predicates a new rule (sub-constraint) is introduced in the code. Hence, the predicate call `subconst(`$x_1, \ldots, x_k$`)` establishes a choice point, effectively creating a disjunction. Figure 35 shows a generic example.

```
subconst(x1,...,xk) :- TP.
subconst(x1,...,xk) :- TQ.
```

Figure 35: The introduced Sub-Constraint for a *Disjunction*.

3. **Negation**
   Let $x_1, \ldots, x_k$ be the list of variables occurring in `P`.

B	s(CASP)
`not P`	`not subconst(`$x_1, \ldots, x_k$`)`

Figure 36: The Translation of *Negation*.

In order to simplify the matter of dealing with negated portions of code, a new sub-constraint is added which is called by its parent predicate with negation. Figure 37 shows a generic example.

```
subconst(x1,...,xk) :- TP.
```

Figure 37: The introduced Sub-Constraint for the case of *Negation*.

4. **Implication**

B	s(CASP)
`P => Q`	resolve as: `not P or Q`

Figure 38: The Translation of *Implication*.

5. **Equivalence**

B	s(CASP)
`P <=> Q`	resolve as: `P => Q & Q => P`

Figure 39: The Translation of *Equivalence*.

6. **Existential quantification**

B	s(CASP)
`#(z).(P & Q)`	T_P, T_Q

Figure 40: The Translation of *Existential Quantification*.

By virtue of s(CASP) operating query-driven, simply the conjunction T_P, T_Q is sufficient for expressing whether there exists some values of z satisfying T_P for which T_Q.

Unfortunately, s(CASP) does not provide any inherent predicate for the universal quantifier. However, by using set comprehensions it is possible to express the universal quantifier manually. This approach is further discussed in Section 2.3.3.

7. **Equality, inequality and comparison operators**
This translation applies to the operators $=, \neq, <, >, \leq, \geq$. Without loss of generality, equality is selected to describe how the aforementioned operators are translated.

At first glance, the translation of `E = F` seems straightforward. However, various precautions are taken to deal with predicate calls and arithmetical evaluation. The B expression `a = f(0)`, intuitively translated to s(CASP), yields a semantically invalid term `A = f(0)`. Hence, the translator analyzes the AST to gather information about E's and F's types of expression in order to produce a semantically correct translation.

The framework distinguishes between four cases:

(a) If E and F express primitive values, then: $T_E = T_F$.

(b) If E is primitive and F is not, then: $T_F,\ T_E = V_F$.

(c) If E is non-primitive and F is primitive, then: $T_E,\ V_E = T_F$.

(d) If E and F are both non-primitive, then: $T_E,\ T_F,\ V_E = V_F$.

As it were, the translator operates with a look-ahead of one to generate appropriate s(CASP) code. Non-primitive B expressions e generate an internal temporary variable, expressed by V_e. This prophylactic approach allows for a sound translation while being independent of analyzing larger portions of the underlying AST.

As mentioned when discussing the translation of B's conjunction, s(CASP) is unable to cope with expressions, for instance `0 < x, x = 1`, where required chunks of code are evaluated at a later point in time by the engine. Hence, the translator employs s(CASP)'s parametric CLP capabilities, more precisely CLP($\neq$) and CLP($\mathbb{Q}$). The operators $\neq, <, >, \leq, \geq$ are translated to `\=, #<, #>, #=<, #>=` respectively. Thus, the CLP backend takes care of all the aforementioned temporal challenges related to arithmetics. However, the operator `\=` also expresses constructive negation besides arithmetical inequality compared to operators `#<, #>, #=<, #>=`, which are solely handled by CLP($\mathbb{Q}$). Consequently, the interpreter decides at runtime whether CLP($\neq$) or CLP($\mathbb{Q}$) is applicable [ACS+18].

2.3.3 Translation of Sets

1. **Empty set**

B	s(CASP)
{}	[]

Figure 41: The Translation of an *Empty Set*.

Since s(CASP) does not support sets inherently, they are expressed by lists. This approach does not clash with confounding lists and sets within s(CASP), as B is not offering a list data structure.

2. **Singleton set**
Similar to equality, for a singleton set of the form `{E}` the translator analyzes `E` to decide whether the expression is primitive. For instance, the B expression `x = {1+1}` cannot be straightforwardly translated to `X = [1+1]`, as in s(CASP) the term `1+1` expresses the compound term `'+'(1,1)` and not its arithmetical evaluation.

   Consequently, the framework distinguishes between two cases:

   (a) If `E` is primitive, then: `Tmp` = $[T_E]$.

   (b) If `E` is non-primitive, then: T_E, `Tmp` = $[V_E]$.

   The freshly introduced internal temporary variable `Tmp` denotes the link to a non-primitive expression. The framework takes care to properly join the temporary variables together with the identifiers and expressions they are meant to be assigned to.

3. **Set enumeration**
A set of arbitrary cardinality of the form `{E, F, ...}` follows the pattern of the singleton set, which is expressed by recursive application.

4. **Ordered pair**
   Definition:
   $E \mapsto F = (E, F)$.

   For an ordered pair `E|->F`, the framework distinguishes between four cases:

   (a) If `E` and `F` express primitive values, then: `Tmp = t(`$\mathtt{T}_E, \mathtt{T}_F$`)`.

   (b) If `E` is primitive and `F` is not, then: $\mathtt{T}_F$, `Tmp = t(`$\mathtt{T}_E, \mathtt{V}_F$`)`.

   (c) If `E` is non-primitive and `F` is primitive, then: $\mathtt{T}_E$, `Tmp = t(`$\mathtt{V}_E, \mathtt{T}_F$`)`.

   (d) If `E` and `F` are both non-primitive, then: $\mathtt{T}_E$, $\mathtt{T}_F$, `Tmp = t(`$\mathtt{V}_E, \mathtt{V}_F$`)`.

5. **Set comprehension**
   Let $x_1, \ldots, x_i$ be all variables occurring in P and let $z = z_1, \ldots, z_j$ be the list of variables constrained by P. Let $y_1, \ldots, y_k$ with $i = j + k$ denote external variables that occur in P but not in z. The set comprehension $\{z|P\}$ is the set of every value of z that satisfies P.

B	**s(CASP)**
`{z|P}`	`subconst1(Tmp,` $y_1, \ldots, y_k$`)`

Figure 42: The Translation of a *Set Comprehension*.

Figure 43 shows how the introduced sub-constraint `subconst1(Tmp,` $y_1, \ldots, y_k$`)` of Figure 42 creates a local scope, where the variables of z, which merely serve the purpose of defining the underlying set, are unable to clash with the variables of the parent scope.

```
subconst1(Tmp, y1,...,yk) :-
 findall(Ψ, subconst2(x1,...,xi), Tmp).
```

Figure 43: The firstly introduced Sub-Constraint for a *Set Comprehension*.

The built-in predicate `findall/3` is used to store in `Tmp` every instance of Ψ, which satisfies `subconst2(`$x_1, \ldots, x_i$`)` expressing $\mathtt{T}_P$. The variable Ψ embodies z, if $|z| = 1$, and an ordered pair of the form $t(z_1, t(z_2, t(\ldots, z_j)))$ otherwise. Finally, Figure 44 shows the translation of P.

```
subconst2(x1,...,xi) :- TP.
```

Figure 44: The secondly introduced Sub-Constraint for a *Set Comprehension*.

6. **Universal quantification**
   Let $z = z_1, \ldots, z_k$ be the list of variables constrained by P. Further, let $p_1, \ldots, p_i$ and $q_1, \ldots, q_j$ be the external variables that occur in P and Q respectively but do not occur in z.

B	s(CASP)
`!(z).(P => Q)`	`subconst1(`$p_1, \ldots, p_i, q_1, \ldots, q_j$`)`

Figure 45: The Translation of the *Universal Quantification*.

Since the built-in `forall/2` predicate unfortunately cannot be invoked in source code, a manual solution is necessary. As this framework is restricted to finite domain declarations, it allows for using set comprehensions to express that Q is true for each value of z satisfying P. Figure 45 depicts the introduced call to the predicate that resolves the universal quantifier. Thereby, the set comprehension $\{z|P\}$ is computed and produces additional code for verifying that Q also holds, illustrated in Figure 46. The variable Ψ again embodies z, if $|z| = 1$, and an ordered pair $t(z_1, t(z_2, t(\ldots, z_k)))$ otherwise.

```
subconst1(p_1,...,p_i,q_1,...,q_j) :-
 findall(Ψ,subconst2(p_1,...,p_i,z_1,...,z_k),Tmp),
 for_all(Tmp,q_1,...,q_j).

subconst2(p_1,...,p_i,z_1,...,z_k) :- T_P.

for_all([],q_1,...,q_j).
for_all([Ψ|Tmp],q_1,...,q_j) :-
 T_Q,
 for_all(Tmp,q_1,...,q_j).
```

Figure 46: The produced Code for the *Universal Quantifier*.

7. **Union**

B	s(CASP)
`S \/ D`	`T`$_S$`, T`$_D$`, union(V`$_S$`,V`$_D$`,Tmp)`

Figure 47: The Translation of a *Union*.

Figure 47 depicts the predicate `union/3`, which expects two ground sets and computes their respective union `Tmp`. Since `S` and `D` are sets and, hence, non-primitive values, the translator resolves them in T_S, T_D to obtain their corresponding links V_S, V_D. In the following, I do not repeatedly emphasize in detail the way custom predicates for non-primitive data are resolved, as it is analogous to how `union/3` is handled. For predicates of arity two, merely the translation of `D` is omitted.

By virtue of restricting this operation to ground instances, limitations are placed on the way of expressing a union. For example, the engine is unable to calculate the result of the expression `S = {1,2} & x = S \/ D & D = {2,3}`. This restraint applies to all of the following

predicates involving non-primitive data structures. In future work this issue could be countered by analyzing the AST so that generated code is solely reliant on already translated expressions.

As posting code of every of the following predicates would yield a too cluttered thesis, the predicates' respective code snippets are included in the `preliminaries.pl` file of this project[5]. Further, the predicates of `preliminaries.pl` are also referred to as *preliminary predicates*.

8. **Intersection**

B	**s(CASP)**
`S /\ D`	$\mathtt{T}_S$, $\mathtt{T}_D$, `inter(`$\mathtt{V}_S$,$\mathtt{V}_D$,`Tmp)`

Figure 48: The Translation of an *Intersection*.

9. **Difference**
Definition:
$S - D = \{x | x \in S \wedge x \notin D\}$.

B	**s(CASP)**
`S-D`	$\mathtt{T}_S$, $\mathtt{T}_D$, `subtract(`$\mathtt{V}_S$,$\mathtt{V}_D$,`Tmp)`

Figure 49: The Translation of a *Difference*.

10. **Cartesian product**
Definition:
$S \times D = \{(x, y) | x \in S \wedge y \in D\}$.

B	**s(CASP)**
`S * D`	$\mathtt{T}_S$, $\mathtt{T}_D$, `cartesian(`$\mathtt{V}_S$,$\mathtt{V}_D$,`Tmp)`

Figure 50: The Translation of a *Cartesian Product*.

11. **Powerset**
Definition:
$\mathbb{P}(S) = \{x | x \subseteq S\}$ and the non-empty subset $\mathbb{P}_1(S) = \mathbb{P}(S) - \{\{\}\}$.

B	**s(CASP)**
`POW(S)`	$\mathtt{T}_S$, `pow(`$\mathtt{V}_S$,`Tmp)`

Figure 51: The Translation of a *Powerset*.

[5] https://gitlab.cs.uni-duesseldorf.de/stups/prob/prob_prolog/-/blob/feature/btoasp/extensions/btoasp/preliminaries.pl

Since this translation considers finite sets only, the translation of finite subsets `FIN(S)` and `FIN1(S)` is the same as `POW(S)` and `POW1(S)` respectively.

12. **Cardinality**
Definition:
$card(S) = |S|$ is defined on finite sets only.

B	s(CASP)
`card(S)`	$\texttt{T}_S$, `card(`$\texttt{V}_S$`,Tmp)`

Figure 52: The Translation of a Set's *Cardinality*.

13. **Set predicates**

Operator	B	s(CASP)
$E \in S$	`E : S`	$\texttt{T}_E$, $\texttt{T}_S$, `member(`$\texttt{V}_E$`,`$\texttt{V}_S$`)`
$E \notin S$	`E /: S`	$\texttt{T}_E$, $\texttt{T}_S$, `not member(`$\texttt{V}_E$`,`$\texttt{V}_S$`)`
$E \subseteq S$	`E <: S`	$\texttt{T}_E$, $\texttt{T}_S$, `subset(`$\texttt{V}_E$`,`$\texttt{V}_S$`)`
$E \not\subseteq S$	`E /<: S`	$\texttt{T}_E$, $\texttt{T}_S$, `not subset(`$\texttt{V}_E$`,`$\texttt{V}_S$`)`
$E \subset S$	`E <<: S`	$\texttt{T}_E$, $\texttt{T}_S$, `strsubset(`$\texttt{V}_E$`,`$\texttt{V}_S$`)`
$E \not\subset S$	`E /<<: S`	$\texttt{T}_E$, $\texttt{T}_S$, `not strsubset(`$\texttt{V}_E$`,`$\texttt{V}_S$`)`

Figure 53: The Translation of *Set Predicates*.

14. **Generalized union**
Definition: $union(U) = \bigcup_{i=1}^{n} S_i$ with $S_i \in U$ and $|U| = n$.

Essentially, the translator stacks successively as many `union/3` calls as necessary to compute the generalized union of the form `union(U)`.

The framework distinguishes between three cases:

(a) If `U = {S}`, then resolve `S` as a singleton set.

(b) If `U = {S, D}`, then treat it as `S \/ D`.

(c) If $n \geq 3$, then resolve $\texttt{S}_1, \texttt{S}_2 \in \texttt{U}$ as $\texttt{S}_1$ `\/` $\texttt{S}_2$ obtaining $\texttt{Tmp}_1$ and continue resolving $\texttt{Tmp}_i$ and the next set $\texttt{S}_{i+2}$ with $i < n - 1$ as

$$\texttt{T}_{Si+2},\ \texttt{union(Tmp}_i\texttt{,V}_{Si+2}\texttt{,Tmp}_{i+1}\texttt{)} \tag{13}$$

until the final result $\texttt{Tmp}_{n-1}$ is computed.

15. **Generalized intersection**
Definition:
$inter(U) = \bigcap_{i=1}^{n} S_i$ with $S_i \in U$ and $|U| = n$.

The code for a general intersection `inter(U)` is generated analogously to how a generalized union is treated.

2.3.4 Translation of Numbers

1. **Minimum and maximum**

B	s(CASP)
`min(S)`	$\mathtt{T}_S$, `min(`$\mathtt{V}_S$`,Tmp)`
`max(S)`	$\mathtt{T}_S$, `max(`$\mathtt{V}_S$`,Tmp)`

Figure 54: The Translation of *Minimum* and *Maximum*.

2. **Arithmetical evaluations**
   This translation applies to the arithmetical operators $+, -, *, /, mod$. Without loss of generality, addition is selected to describe how the aforementioned operators are translated. CLP($\mathbb{Q}$) is employed to handle arithmetical operations, thereby omitting temporal restraints of arithmetical evaluation in s(CASP). The CLP($\mathbb{Q}$) counterpart `#=/2`, which subsumes and extends `is/2`, is used instead. However, the Euclidean division is in s(CASP) expressed by `A is mod(m,n)`, thus restricting `m` and `n` to be ground prior to its call. The translator examines the ASTs of B expressions `m` and `n` and provides their respective evaluations.

   The framework distinguishes between four cases for `m+n`:

   (a) If `m` and `n` express primitive values, then: `Tmp #=` $\mathtt{T}_m$+$\mathtt{T}_n$.

   (b) If `m` is primitive and `n` is not, then: $\mathtt{T}_n$, `Tmp #=` $\mathtt{T}_m$+$\mathtt{V}_n$.

   (c) If `m` is non-primitive and `n` is primitive, then: $\mathtt{T}_m$, `Tmp #=` $\mathtt{V}_m$+$\mathtt{T}_n$.

   (d) If `m` and `n` are both non-primitive, then: $\mathtt{T}_m$, $\mathtt{T}_n$, `Tmp #=` $\mathtt{V}_m$+$\mathtt{V}_n$.

3. **Interval**
   Definition:
   $m..n = \{i | m \leq i \leq n\}$

   The framework distinguishes between two cases in order to enhance performance:

   (a) If `m,n` are primitive, then let the framework directly compute: `Tmp` = $[\mathtt{T}_m, \ldots, \mathtt{T}_n]$.

   (b) If `m,n` are non-primitive, then: $\mathtt{T}_m$, $\mathtt{T}_n$, `range(`$\mathtt{V}_m$, $\mathtt{V}_n$, `Tmp)`.

2.3.5 Translation of Relations

A relation r is a set of ordered pairs.

1. **Relations**
   Definition:
   $S \leftrightarrow D = \mathbb{P}(S \times D)$

B	s(CASP)
`S <-> D`	$\texttt{T}_S$, $\texttt{T}_D$, `relation(`$\texttt{V}_S$,$\texttt{V}_D$`,Tmp)`

Figure 55: The Translation of a set of *Relations*.

2. **Domain and range**
   Definition of a domain:
   $dom(r) = \{x | \exists y : x \mapsto y \in r\}$

   Definition of a range:
   $ran(r) = \{y | \exists x : x \mapsto y \in r\}$

B	s(CASP)
`dom(r)`	$\texttt{T}_r$, `dom(`$\texttt{V}_r$`,Tmp)`
`ran(r)`	$\texttt{T}_r$, `ran(`$\texttt{V}_r$`,Tmp)`

Figure 56: The Translation of a *Domain* and *Range*.

3. **Composition**
   Definition:
   $r_1; r_2 = \{(x, y) | \exists z : x \mapsto z \in r_1 \land z \mapsto y \in r_2\}$

B	s(CASP)
`r1;r2`	$\texttt{T}_{r1}$, $\texttt{T}_{r2}$, `comp(`$\texttt{V}_{r1}$,$\texttt{V}_{r2}$`,Tmp)`

Figure 57: The Translation of a *Composition*.

4. **Identity**
   Definition:
   $id(S) = \{(x, x) | x \in S\}$

B	s(CASP)
`id(S)`	$\texttt{T}_S$, `id(`$\texttt{V}_S$`,Tmp)`

Figure 58: The Translation of the *Identity*.

5. **Restriction and subtraction**
   Definition of a domain restriction:
   $S \lhd r = \{(x, y) | x \mapsto y \in r \land x \in S\}$

   Definition of a domain subtraction:
   $S \lhd\!\!\!\!- r = \{(x, y) | x \mapsto y \in r \land x \notin S\}$

   Definition of a range restriction:
   $r \rhd D = \{(x, y) | x \mapsto y \in r \land y \in D\}$

   Definition of a range subtraction:
   $r \rhd\!\!\!\!- D = \{(x, y) | x \mapsto y \in r \land y \notin D\}$

Operator	**B**	**s(CASP)**
$S \lhd r$	`S <| r`	T_S, T_r, `domres(`V_S`,`V_r`,Tmp)`
$S \lhd\!\!\!\!- r$	`S <<| r`	T_S, T_r, `domsub(`V_S`,`V_r`,Tmp)`
$r \rhd D$	`r |> D`	T_S, T_r, `ranres(`V_S`,`V_r`,Tmp)`
$r \rhd\!\!\!\!- D$	`r |>> D`	T_S, T_r, `ransub(`V_S`,`V_r`,Tmp)`

Figure 59: The Translation of a *Restriction* and a *Subtraction* for *Domains* and *Ranges*.

6. **Inverse**
   Definition:
   $r^{-1} = \{(y, x) | x \mapsto y \in r\}$

B	**s(CASP)**
`r~`	T_r, `inverse(`V_r`,Tmp)`

Figure 60: The Translation of an *Inverse* Relation.

7. **Relational image**
   Definition:
   $r[S] = \{y | \exists x : x \in S \land x \mapsto y \in r\}$

B	**s(CASP)**
`r[S]`	T_S, T_r, `image(`V_S`,`V_r`,Tmp)`

Figure 61: The Translation of an *Relational Image*.

8. **Overriding**
   Definition:
   $r_1 \lhd\!\!\!\!+ r_2 = r_2 \cup (dom(r_2) \lhd\!\!\!\!- r_1)$

B	s(CASP)
`r1 <+ r2`	$\mathtt{T}_{r1}$, $\mathtt{T}_{r2}$, $\mathtt{override}(\mathtt{V}_{r1}, \mathtt{V}_{r2}, \mathtt{Tmp})$

Figure 62: The Translation of *Overriding*.

9. **Direct product**
   Definition:
   $r_1 \otimes r_2 = \{(x,(y,z))|x \mapsto y \in r_1 \wedge x \mapsto z \in r_2\}$

B	s(CASP)
`r1 >< r2`	$\mathtt{T}_{r1}$, $\mathtt{T}_{r2}$, $\mathtt{dpro}(\mathtt{V}_{r1}, \mathtt{V}_{r2}, \mathtt{Tmp})$

Figure 63: The Translation of a *Direct Product*.

10. **Parallel product**
    Definition:
    $r_1 \parallel r_2 = \{(x,y),(m,n)|x \mapsto m \in r_1 \wedge y \mapsto n \in rca\}$

B	s(CASP)
`r1 || r2`	$\mathtt{T}_{r1}$, $\mathtt{T}_{r2}$, $\mathtt{ppro}(\mathtt{V}_{r1}, \mathtt{V}_{r2}, \mathtt{Tmp})$

Figure 64: The Translation of a *Parallel Product*.

2.3.6 Translation of Functions

A function f is a relation for which holds that for each element x of f's domain there is exactly one ordered pair $x \rightarrow y \in f$.

1. **Partial functions**
   Definition :
   $S \nrightarrow D = \{r|r \in S \leftrightarrow D \wedge r^{-1}; r \subseteq id(D)\}$

B	s(CASP)
`S +-> D`	$\mathtt{T}_S$, $\mathtt{T}_D$, $\mathtt{pfun}(\mathtt{V}_S, \mathtt{V}_D, \mathtt{Tmp})$

Figure 65: The Translation of *Partial Functions*.

2. **Total functions**
   Definition :
   $S \rightarrow D = \{f|f \in S \nrightarrow D \wedge dom(f) = S)\}$

B	s(CASP)
`S --> D`	$\texttt{T}_S$, $\texttt{T}_D$, `tfun(`$\texttt{V}_S$,$\texttt{V}_D$,`Tmp)`

Figure 66: The Translation of *Total Functions*.

3. **Partial injections**
   Definition :
   $S \rightarrowtail\!\!\!\!\!\!\rightarrow D = \{f | f \in S \nrightarrow D \wedge f^{-1} \in D \nrightarrow S\}$

B	s(CASP)
`S >+> D`	$\texttt{T}_S$, $\texttt{T}_D$, `pinj(`$\texttt{V}_S$,$\texttt{V}_D$,`Tmp)`

Figure 67: The Translation of *Partial Injections*.

4. **Total injections**
   Definition :
   $S \rightarrowtail D = S \rightarrowtail\!\!\!\!\!\!\rightarrow D \cap S \rightarrow D$

B	s(CASP)
`S >-> D`	$\texttt{T}_S$, $\texttt{T}_D$, `tinj(`$\texttt{V}_S$,$\texttt{V}_D$,`Tmp)`

Figure 68: The Translation of *Total Injections*.

5. **Partial surjections**
   Definition :
   $S \twoheadrightarrow D = \{f | f \in S \nrightarrow D \wedge ran(f) = D\}$

B	s(CASP)
`S +->> D`	$\texttt{T}_S$, $\texttt{T}_D$, `psur(`$\texttt{V}_S$,$\texttt{V}_D$,`Tmp)`

Figure 69: The Translation of *Partial Surjections*.

6. **Total surjections**
   Definition :
   $S \twoheadrightarrow D = S \twoheadrightarrow\!\!\!\!\!\!\rightarrow D \cap S \rightarrow D$

B	s(CASP)
`S -->> D`	$\texttt{T}_S$, $\texttt{T}_D$, `tsur(`$\texttt{V}_S$,$\texttt{V}_D$,`Tmp)`

Figure 70: The Translation of *Total Surjections*.

7. **Bijections**
   Definition :
   $S \rightarrowtail\!\!\!\!\!\!\twoheadrightarrow D = S \rightarrowtail D \cap S \twoheadrightarrow D$

B	s(CASP)
`S >->> D`	$\texttt{T}_S$, $\texttt{T}_D$, `tbij(`$\texttt{V}_S$,$\texttt{V}_D$,`Tmp)`

Figure 71: The Translation of *Bijections*.

8. **Function application**
   Definition :
   $E \mapsto y \in f \Rightarrow f(E) = y$

Formal	B	s(CASP)
$f(E)$	`f(E)`	T_E, T_f, `fun(`Tmp_f, Tmp_E, Tmp`)`

Figure 72: The Translation of a *Function Application*.

Note that the framework expects a call for `fun/3` to be well-defined.

3 Empirical Evaluation

In this section I aim towards empirically evaluating the presented framework's capabilities by evaluating its correctness and comparing benchmark performances of s(CASP) to the native ProB, Kodkod and Z3 backend. Furthermore as s(CASP) is a novel execution model and, as the authors state, lacks a well rounded evaluation via benchmarking itself, this examination serves as a closer analysis of s(CASP) as well [ACS+18]. Foremost, in Section 3.1 the framework's correctness is investigated. Section 3.2 focuses on runtime performance of individual predicates using artificial benchmarks. Further, in Section 3.3 the computation of answers for variables bound to constructive disequalities is investigated. Finally, Section 3.4 concludes with the evaluation of some real-world examples.

3.1 Evaluation of Correctness

As this thesis implements a constraint solving backend, it is naturally imperative to ensure its correctness. Therefore, test cases[6] are embedded in ProB consisting of numerous exemplary expressions, which cover the supported predicates. For each test, the s(CASP) backend first solves the underlying predicate obtaining a result, which is afterwards validated by ProB. Of course, this evaluation does not prove the backend's correctness. However, this allows for a better insight into its possible flaws and aids further development.

[6] https://gitlab.cs.uni-duesseldorf.de/stups/prob/prob_prolog/-/blob/feature/btoasp/src/testcases.pl

3.2 Performance Evaluation of Individual Predicates

This evaluation is split into three parts. I focus on evaluating the performance of predicates for sets in Section 3.2.1, relations in Section 3.2.2 and functions in Section 3.2.3. The evaluation of primitive values, numbers and logic expressions is covered in Section 3.4, as they appear on a regular basis in real-world problems. Furthermore as sets, relations and functions are build upon each other, the evaluation of the latter predicates indirectly includes the prior ones.

In the following, I present tables showing the runtime performances for each predicate of the aforementioned predicates classes. The empirical result for a predicate is gathered by solving and measuring the runtime of three separate synthetic B expressions on each backend (if available) via the ProB REPL. Additionally, the runtime of the plain s(CASP) engine is measured to gain more insight into s(CASP)'s performance by omitting the surrounding overhead introduced by the framework's processes. Each computation is executed three times with a one minute threshold on a freshly initialized REPL to counter inaccuracies in the measurements. The environment of this evaluation is a macOS 10.14.6 machine operating on a 7th generation i5 Intel processor at 3.1GHz. The displayed value representing a predicate's performance in milliseconds is obtained by averaging over the results of all three exemplary expressions where every individual value is rounded half away from zero. The designation `n.a.` indicates that either a timeout or an *unsupported predicate exception* occurred. The benchmarks along with all performance measures for each run can be found on GitLab[7].

3.2.1 Performance Evaluation of Sets

Predicate	Native ProB	Kodkod	Z3	s(CASP)	Plain s(CASP)
Set Comp.	78	39	`n.a.`	263	16
Univ. Quantifier	96	40	82	257	2
Union	67	53	101	310	47
Intersection	70	52	103	323	38
Difference	72	54	108	319	42
Cart. Product	76	40	`n.a.`	293	5
Powerset	74	`n.a.`	`n.a.`	1820	88
Cardinality	50	`n.a.`	`n.a.`	241	1
Gen. Union	55	42	94	302	52
Gen. Inter.	56	50	99	311	61

Figure 73: Comparison of Set Predicate Performances measured in Milliseconds.

The table of Figure 73 indicates at first glance that the s(CASP) backend performs rather poorly compared to the other three employed backends, as its runtimes are noticeably higher. The pred-

[7] https://gitlab.cs.uni-duesseldorf.de/efraimidis/b-to-asp/-/blob/master/thesis/MA/eval.md

icates union, intersection, difference, powerset, general union and general intersection consume visibly more time than the remaining ones. I noticed that particularly the execution time of the largest expression of the powerset benchmarks is significantly extensive compared to the other ones. This may be caused by the incorporation of the built-in `findall/3` predicate for computing a powerset. The rest of the aforementioned predicates strikingly often invoke `member/2`. As s(CASP) lacks a cut operator contrary to Prolog, additional rules containing the call `not member/2` are needed to correctly express the predicate's behavior. Consequently, in the instance of an element not being a member of a list the dual rule resolving the negated goal is called nevertheless. This may induce further loss of time, especially for large lists.

However, s(CASP) is able to obtain a result in some cases where Kodkod and Z3 are unable to follow. Furthermore, the performances of executing the plain s(CASP) engine seem to be reasonable, as the corresponding runtimes are noticeably low. Especially the predicates set comprehension, universal quantifier, cartesian product and cardinality stand out. Cartesian product and cardinality essentially iterate over a list without having to compute any other complex side task. On the other hand, set comprehension and universal quantifier work similarly but are also dependent on the `findall/3` constraint's complexity. Therefore, I assume that the specific constraints stated in these benchmarks are not particularly challenging. Yet, the respective runtimes for the s(CASP) backend are perceptibly slower compared to the other ones. This occurrence suggests that the framework's overhead is considerably prevalent.

Figure 74 depicts a distribution of time for a solve process within the ProB REPL for the s(CASP) backend. This data is obtained by computing the sum of the averaged runtimes for each sub task of the largest expressions of the benchmarks for the predicates that are shown in Figure 73. Note that these percentages merely pose an estimate on how the framework performs for any given expression.

Total	**Trans.**	**File**	**Init. Inter.**	**Exec. Inter.**	**Solve**	**P. Process**	**Rest**
8527	104	12	10	2387	5659	93	262
100%	1.3%	0.1%	0.1%	28%	66.4%	1%	3.1%

Figure 74: Time Distribution of the s(CASP) Solve Process measured in Milliseconds.

According to the data of Figure 74, the framework spends 28% of the time communicating with SICStus and Ciao Prolog via the interface, which is a considerable amount relative to the actual solving time of 66.4%. This loss of time is caused by having to start the Ciao engine, which consequently initializes the s(CASP) interpreter before a solution can be computed. On my machine I recorded an average startup time of 130ms for the Ciao engine. In perspective of the times regarding s(CASP) displayed in Figure 73, the time of initialization renders roughly half of the entire process. However, these benchmarks are rather small compared to real-world examples. Hence, the time loss of the Ciao engine's initialization is here more apparent. Note that the spike of the powerset's runtime lowers the overall percentage of the interface's workload, as the startup time of the Ciao engine remains constant irrespective of the given problem. Nevertheless, this poses an

unsatisfying persistent weakness of the framework.

The percentages of the remaining tasks, i.e. translation, file generation, post-processing and the internal processes of ProB , referred to as *rest*, are not standing out compared to the other ones. However, this data is insufficient to judge whether the runtimes of the aforementioned tasks remain unnoticeable for any other arbitrary predicate.

3.2.2 Performance Evaluation of Relations

Predicate	Native ProB	Kodkod	Z3	s(CASP)	Plain s(CASP)
Relations	94	n.a.	n.a.	698	39
Domain/Range	55	31	132	260	15
Composition	55	30	n.a.	245	1
Identity	67	n.a.	n.a.	249	0
Restrict/Subtract	54	30	98	256	14
Inverse	54	41	89	244	0
Rel. Image	56	42	102	245	4
Override	54	n.a.	n.a.	7639	7327
Direct Product	54	n.a.	n.a.	249	6
Parallel Prod.	55	n.a.	n.a.	280	12

Figure 75: Comparison of Relation Performances measured in Milliseconds.

The results of Figure 75 are generally reminiscent of the well performing predicates regarding sets. The predicates composition, identity, inverse, relational image, direct product and parallel product are noticeably efficient due to being facile in the sense that they all merely iterate over a list without any demanding sub task along the way. On the other hand, the predicates domain, range, domain/range restriction/subtraction and override are using `member/2` of which the entirety, except for override, also performs solidly. However, the presence of the framework's overhead somewhat discards the gained efficiency.

Interestingly, the largest expression of the override benchmarks consumes a significantly substantial amount of time, which leads to the spike of 7639ms. Since override calls the domain, domain subtraction and union predicate, which all invoke `member/2`, the earlier assumption that this could prove problematic for large lists is supported. Nevertheless, the other ones perform solidly, which could be due to coincidentally advantageous benchmarks or vice versa in the case of override.

The computation of a set of relations requires to compute the powerset of the underlying cartesian product. Hence, the runtime of the relations predicate presumably suffers from the poor performance of the powerset computation. Yet, for these benchmarks the performance of the plain engine is acceptable.

3.2.3 Performance Evaluation of Functions

Predicate	Native ProB	s(CASP)	Plain s(CASP)
Partial Functions	76	885*	333*
Total Functions	75	414*	161*
Partial Injections	83	537*	311*
Total Injections	77	1639*	1354*
Partial Surjections	77	379*	142*
Total Surjections	78	496*	256*
Bijections	76	2402*	2246*
Function Application	54	241	0

Figure 76: Comparison of Function Performances measured in Milliseconds.

The performance measurements for functions of Figure 76 emphasize an overall weak performance for the s(CASP) backend. The asterisk indicates that the largest expression of the benchmarks is not succeeding for the given threshold of one minute. Therefore, the displayed values express just the average of the two smaller expressions, including ProB. Furthermore, it seems that the plain computation time is also underwhelming in contrast to the majority of the previously covered predicates. The comparison of ProB to s(CASP) alone is sufficient to demonstrate that s(CASP) is clearly underperforming, thus is not posing a sensible option for computing functions. However, the function application predicate appears to perform efficiently, which is merely a `member/2` call.

The rest of the predicates rely on `findall/3`. This supports the assumption of Section 3.2.1 and Section 3.2.2 that similar to the computation of a powerset the deployment of `findall/3` poses a considerable bottleneck.

Certainly, the implementation of the preliminary predicates can be enhanced. However, the framework offers a run option called *optimize*. A ground B expression, `x = {1,2,3} --> {4,5,6}` for instance, is then computed by ProB before being passed to the framework. In this case, the resolved set of total functions is translated to ground s(CASP) code, which prevents an inferior computation by the s(CASP) engine.

Moreover, as s(CASP) is not employing laziness the engine tries to compute a set of functions regardless of its use and size contrary to ProB. Thus, even given a more sophisticated implementation of preliminary predicates without the involvement of `findall/3`, the process would presumably still lack efficiency in some cases.

3.3 Performance Evaluation of Constructive Negation

In this subsection I analyze the efficiency of computing an answer out of a constructive negation term. Since the aim is to focus on the post processing, I evaluate the runtimes of ground inequali-

ties. Note that the evaluation of the Kodkod backend is omitted, as it does not offer this feature.

Native ProB	Z3	s(CASP)	Post Processing
67	84*	252	2

Figure 77: Comparison of Generating Answers out of Constructive Negation Terms Performances measured in Milliseconds.

The table of Figure 77 expresses the average runtimes for the three backends to return a result for exemplary inequalities. The data is gathered by averaging over different types of expressions. Inequalities of integers, ordered pairs, sets, sets of ordered pairs and sets of sets are considered in this analysis. The asterisk indicates that the Z3 backend is unable to compute an answer for one of the benchmarks.

Generally, the average time of 2ms resolving constructive negation terms with s(CASP) seems to be acceptable. However, regardless of the actual solving time of 2ms the framework's overhead is still present. Furthermore, this evaluation includes structures containing at least one integer only, as data without integers is not supported. Therefore, one needs to consider the s(CASP) backend's limitations.

3.4 Performance Evaluation of Real-World Examples

In the following, I evaluate the s(CASP) backend by measuring performances of real-world examples.

Predicate	N. ProB	Kodkod	Z3	s(CASP)	P. s(CASP)	Translation
4-Queens	115	36	126	369	31	78
5-Queens	112	36	151	438	37	85
6-Queens	117	37	314	4927	3101	225
7-Queens	107	39	985	30507	5190	7092

Figure 78: Comparison of N-Queens Performances measured in Milliseconds.

Figure 78 shows the performances of various ProB backends dealing with different sizes of the *n-queens* problem[8]. Note that the performance measurements of s(CASP) are supported by ProB. That is, the tool makes use of the run option *optimize*, hence ground instances are computed by ProB first. The reason is that this implementation seems to struggle with the computation of functions. Consequently, the raw evaluation of the aforementioned predicates does not succeed in the predefined time window of one minute. Furthermore, the s(CASP) backend is unfortunately unable

[8] https://gitlab.cs.uni-duesseldorf.de/stups/prob/prob_examples/-/blob/master/public_examples/Eval/NQueens.eval

to resolve other real-world examples, such as *sudoku*, *graph coloring* and *who killed Agatha*[9], as they are all defined using functions. Therefore, they are not included in the table of Figure 78.

The s(CASP) backend is slower than the other solvers for $n = 4$ and $n = 5$, however it still seems to perform quite reasonable. Interestingly, the runtimes rise very rapidly for an increasing amount of queens. Especially the recorded time for $n = 7$ appears to be extraordinary high. As the evaluated set of total functions for `1..7 >-> 1..7` has to be printed entirely, it causes the translator to generate a large portion of extra code resulting in total time of 7092ms for the task. Though, combined with the actual solving time of 5190ms there is a close to 20 second time gap left. This is due to having to compute and generate a stable model, which is extremely large compared to the smaller problem instances, and having to synthesize a dual program before the query can be resolved. For this problem, I recorded an average time of roughly four seconds after invoking the engine and before the query was resolved. This suggests that for especially large problem instances perhaps the initial computation of the dual program might be considerable as well.

One needs to consider that this translation hardly incorporates any optimizations. Predicates are in that sense straightforwardly translated. Therefore, it is plausible that the backend's performance is also dependent on how suitable the problem definition is for the s(CASP) methodology. However, this is subject to future work.

Overall, the results are underwhelming. The s(CASP) backend seems to perform passable for smaller predicates. Nevertheless, for larger instances the backend generally experiences considerable difficulties to obtain an answer. In particular, the s(CASP) backend is outclassed by the native ProB, Kodkod and Z3 backend for every single benchmark. In some few cases, however, Kodkod and Z3 are unable to find a solution where s(CASP) succeeds. Yet, even in those instances native ProB poses a superior option. Thus, the s(CASP) backend is hardly recommendable. The gathered results suggest that parts of the preliminary predicates are underperforming due to invoking `findall/3` as well as the s(CASP) engine itself. By virtue of having to unnecessarily resolve some predicate's respective negated call owing to the cut operator's absence, the solver clearly lacks efficiency in several cases. However, with sophisticated refinement this backend could be rendered more beneficial.

4 Future Work

Completeness Even though the presented framework is capable of translating a large portion of the B realm to s(CASP), there are some predicates left to be supported. Predicates such as iteration, closure, projection, lambda abstraction and the coverage of sequences should be included for broadening the range of legal predicates to be translated. Enabling one to assign variables to infinite domains, for example the B expression `n:NAT`, is decisive to be able to express more complex problems. As s(CASP) offers a CLP backend and shares similarities with Prolog, this probably

[9] https://gitlab.cs.uni-duesseldorf.de/stups/prob/prob_examples/-/blob/master/public_examples/Eval

could be done similarly to how ProB handles those instances [KL17]. Further, the approach of computing answers out of constructive negation terms is unable to handle expressions without integers. Since ProB is capable of coping with resolving inequalities, it should be possible to let ProB handle this task. Furthermore, the framework could be enhanced to deal with predicates of the form `a = {1,2} & x = a \/ b & b = {2,3}`, where variables are used prior to their declaration. Arithmetical instances of the form `a = b+1 & b = 1` are covered by s(CASP)'s CLP($\mathbb{Q}$) solver. However, this feature should be enabled for all predicates. This could be done by analyzing the underlying AST within the framework, and thus generate appropriate code that is solely reliant on already evaluated data.

Efficiency Considering the empirical evaluation's results of Section 3, it is incontrovertible that the presented backend is not on par with the other ones employed by ProB. The translation process can certainly be enhanced as for very large programs a considerable amount of time is consumed. That is, the generated program containing a set of over 5000 ordered pairs for the evaluated expression `1..7 >-> 1..7` took significantly longer than generating code for the smaller *n-queens* instances. The translator gradually generates a list of atoms when walking over the AST. Hence, the framework could gain performance by expressing the s(CASP) code as a difference list of atoms to escape linearity when concatenating. Reimplementing parts of the preliminary predicates that utilize `findall/3` in a more efficient way should also lead to a noticeable performance boost. Additionally, research could be done on how problem instances are properly defined in order to be solved more efficiently. Furthermore, the evaluation of Section 3.2.2 suggests that s(CASP) itself seems to be immature to deal with some problem instances due to the involvement of dual rules. The engine is under continuous development lead by Arias and Carro. During my work, I encountered a few bugs and purposed some suggestions to enhance parts of s(CASP), which were incorporated into the engine. Yet, considering the current state of s(CASP) it seems sensible to explore further options.

Alternatives This work presents a framework translating B predicates to s(CASP), which is an implementation of the Answer Set Programming formalism. However, there are numerous other dialects of ASP available, which are covered in Section 1.4.2. Perhaps choosing a classical ASP implementation may lead to more promising performances. On the other hand, the grounding bottleneck is then to consider [BLS13]. Furthermore, opting for a classical approach would erase the freedom regarding variables' domains that s(CASP) is offering.

More sophisticated performance evaluation Lastly, this work could be incorporated in the analysis of automatically selecting a suitable backend for ProB, researched by Dunkelau [Dun17]. Thereby, methods of deep learning decide which backend is most suitable for the underlying problem [LBH15, GBCB16]. Considering this empirical evaluation, such research is presumably more sensible for an already enhanced and better performing backend.

5 Conclusion

In this thesis I developed an additional constraint solving backend for ProB, which translates B predicates to s(CASP). The presented framework enables predicates to be solved by the s(CASP) engine within ProB via the implemented Ciao/SICStus Prolog interface. Special attention has been paid to the framework's design and implementation and to evaluate its performance by comparing it to the native, Kodkod and Z3 backend of ProB.

In conclusion, this work poses a foundation for an s(CASP) backend. In particular, the framework is capable of translating a considerable amount of B predicates to s(CASP). However, this work's empirical results of Section 3 indicate weaknesses of the implemented backend for some predicates, especially for functions. Therefore, I suggest to build upon the proposed work. Particularly, the implementation of the preliminary predicates are to be enhanced in order to gain more performance. As the computation of functions seems to be the main bottleneck, I assume that improving their implementation would yield the most notable results. Concurrently, the empirical evaluation shows that the plain s(CASP) engine performs noticeably well for some of the benchmarks, which do not incorporate `findall/3`. Specifically, efficient runtimes have been recorded for the predicates cartesian product, cardinality, composition, identity, inverse, relational image, direct product and function application, which all succeeded on average in less than ten milliseconds. This evaluation indicates that improving the framework as a whole to reduce its surrounding overhead may also lead to better performances and to a more promising backend in general. Furthermore, the presented backend can be used for verification of other employed solvers. Similar to how the s(CASP) backend is tested by ProB, ProB's and other solver's answers could thus be verified by s(CASP). Especially, for predicates that are solely supported by the native backend, the s(CASP) backend can be applied. For instance, the Kodkod and Z3 backend do not support predicates such as direct and parallel product, which can be solved by s(CASP). Yet, this kind of employment merely poses a sensible option in cases where the underlying predicate is solvable by the s(CASP) engine in a reasonable amount of time.

In retrospective, I would have approached this work oppositely. Instead of focusing on translating a large portion of B predicates, I could have aimed towards a more sophisticated and optimized framework. However, this presumably would have induced a more narrow backend regarding supported predicates. Nevertheless, I am of the opinion that prioritizing a more efficient solver would have yielded more satisfying results.

Overall, I consider Answer Set Programming, and more specifically s(CASP), to have potential to improve the field of formal methods and think that this thesis constitutes the foundation towards this direction.

List of Figures

List of Algorithms

References

[Abr96] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 2010.

[ACCG20] Joaquín Arias, Manuel Carro, Zhuo Chen, and Gopal Gupta. Justifications for goal-directed constraint answer set programming. *arXiv preprint arXiv:2009.10238*, 2020.

[ACS+18] Joaquin Arias, Manuel Carro, Elmer Salazar, Kyle Marple, and Gopal Gupta. Constraint answer set programming without grounding. *Theory and Practice of Logic Programming*, 18(3-4):337–354, 2018.

[AD94] Krzysztof R Apt and Kees Doets. A new definition of sldnf-resolution. *The Journal of Logic Programming*, 18(2):177–190, 1994.

[AGL+05a] Christian Anger, Martin Gebser, Thomas Linke, André Neumann, and Torsten Schaub. The nomore++ approach to answer set solving. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 95–109. Springer, 2005.

[AGL+05b] Christian Anger, Martin Gebser, Thomas Linke, André Neumann, and Torsten Schaub. The nomore++ system. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 422–426. Springer, 2005.

[APS04] José Júlio Alferes, Luís Moniz Pereira, and Terrance Swift. Abduction in well-founded semantics and generalized stable models via tabled dual programs. *Theory and Practice of Logic Programming*, 4(4):383–428, 2004.

[AVE82] Krzysztof R Apt and Maarten H Van Emden. Contributions to the theory of logic programming. *Journal of the ACM (JACM)*, 29(3):841–862, 1982.

[BCC+97] Francisco Bueno, Daniel Cabeza, Manuel Carro, Manuel Hermenegildo, P López-Garcıa, and Germán Puebla. The ciao prolog system. *Reference Manual. The Ciao System Documentation Series–TR CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM)*, 95:96, 1997.

[BCC+03] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. 2003.

[BET11] Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011.

[BLS13] Marcello Balduccini, Yuliya Lierler, and Peter Schüller. Prolog and asp inference under one roof. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 148–160. Springer, 2013.

[Bör03] Egon Börger. The asm refinement method. *Formal aspects of computing*, 15(2):237–257, 2003.

[BST+10] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, England)*, volume 13, page 14, 2010.

[BW12] Ralph-Johan Back and Joakim Wright. *Refinement calculus: a systematic introduction*. Springer Science & Business Media, 2012.

[CES86] Edmund M. Clarke, E Allen Emerson, and A Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[CGMR16] Francesco Calimeri, Martin Gebser, Marco Maratea, and Francesco Ricca. Design and results of the fifth answer set programming competition. *Artificial Intelligence*, 231:151–181, 2016.

[CJGK+18] Edmund M Clarke Jr, Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. *Model checking*. MIT press, 2018.

[CL89] Lawrence Cavedon and John W. Lloyd. A completeness theorem for sldnf resolution. *The Journal of Logic Programming*, 7(3):177–191, 1989.

[Cla78] Keith L Clark. Negation as failure. In *Logic and data bases*, pages 293–322. Springer, 1978.

[CM03] Dominique Cansell and Dominique Méry. Foundations of the b method. *Computing and informatics*, 22(3/4):221–256, 2003.

[COC97] Mats Carlsson, Greger Ottosson, and Björn Carlson. An open-ended finite domain constraint solver. In *International Symposium on Programming Language Implementation and Logic Programming*, pages 191–206. Springer, 1997.

[CR93] Alain Colmerauer and Philippe Roussel. The birth of Prolog. In *HOPL-II: The second ACM SIGPLAN conference on the history of programming languages*, pages 37–52, New York, NY, USA, 1993. ACM Press.

[CWA+88] Mats Carlsson, Johan Widen, Johan Andersson, Stefan Andersson, Kent Boortz, Hans Nilsson, and Thomas Sjöland. *SICStus Prolog User's Manual*, volume 3. Swedish Institute of Computer Science, Kista, Sweden, 1988.

[DKS19] Jannik Dunkelau, Sebastian Krings, and Joshua Schmidt. Automated backend selection for prob using deep learning. In *NASA Formal Methods Symposium*, pages 130–147. Springer, 2019.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.

[DPDPR09] Alessandro Dal Palu, Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi. Gasp: Answer set programming with lazy grounding. *Fundamenta Informaticae*, 96(3):297–322, 2009.

[DS06] Nick Drummond and Rob Shearer. The open world assumption. In *eSI Workshop: The Closed World of Databases meets the Open World of the Semantic Web*, volume 15, 2006.

[DTEF+12] Minh Dao-Tran, Thomas Eiter, Michael Fink, Gerald Weidinger, and Antonius Weinzierl. Omiga: An open minded grounding on-the-fly answer set solver. In *European Workshop on Logics in Artificial Intelligence*, pages 480–483. Springer, 2012.

[Dun17] Jannik Dunkelau. Automatic selection of solvers using deep learning. Master's thesis, Heinrich-Heine-University, Duesseldorf, Germany, 2017.

[FBHL73] Abraham Adolf Fraenkel, Yehoshua Bar-Hillel, and Azriel Levy. *Foundations of set theory*. Elsevier, 1973.

[Fer05] Paolo Ferraris. Answer sets for propositional theories. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 119–131. Springer, 2005.

[FFS+18] Andreas Falkner, Gerhard Friedrich, Konstantin Schekotihin, Richard Taupe, and Erich C Teppan. Industrial applications of answer set programming. *KI-Künstliche Intelligenz*, 32(2):165–176, 2018.

[Fit92] Melvin Fitting. The stable model semantics for logic programming, 1992.

[GBCB16] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[GHN+04] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Dpll (t): Fast decision procedures. In *International Conference on Computer Aided Verification*, pages 175–188. Springer, 2004.

[Gho00] Aditya Ghose. Formal methods for requirements engineering. In *Proceedings International Symposium on Multimedia Software Engineering*, pages 13–13. IEEE, 2000.

[GKK+08] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. A user's guide to gringo, clasp, clingo, and iclingo. 2008.

[GKNS07] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. clasp: A conflict-driven answer set solver. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 260–265. Springer, 2007.

[GKSS08] Carla P Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. Satisfiability solvers. *Foundations of Artificial Intelligence*, 3:89–134, 2008.

[GST07] Martin Gebser, Torsten Schaub, and Sven Thiele. Gringo: A new grounder for answer set programming. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 266–271. Springer, 2007.

[Hoa05] Thai Son Hoang. *The development of a probabilistic B-method and a supporting toolkit*. PhD thesis, University of New South Wales, 2005.

[Hol95] Christian Holzbaur. Ofai clp (q, r) manual. Technical report, edition 1.3. 3. Technical Report TR-95-09, Austrian Research Institute for . . . , 1995.

[Hor51] Alfred Horn. On sentences which are true of direct unions of algebras. *The Journal of Symbolic Logic*, 16(1):14–21, 1951.

[Jac02] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[JL87] Joxan Jaffar and J-L Lassez. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 111–119, 1987.

[JN04] Tomi Janhunen and Ilkka Niemelä. Gnt—a solver for disjunctive logic programs. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 331–335. Springer, 2004.

[JSS01] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A micromodularity mechanism. *ACM SIGSOFT Software Engineering Notes*, 26(5):62–73, 2001.

[KBL14] Sebastian Krings, Jens Bendisposto, and Michael Leuschel. Turning failure into proof: evaluating the prob disprover. In *1st International Workshop about Sets and Tools (SETS 2014)*, page 46, 2014.

[KL16] Sebastian Krings and Michael Leuschel. Smt solvers for validation of b and event-b models. In *International Conference on Integrated Formal Methods*, pages 361–375. Springer, 2016.

[KL17] Sebastian Krings and Michael Leuschel. Constraint logic programming over infinite domains with an application to proof. *arXiv preprint arXiv:1701.00629*, 2017.

[Kow74] Robert Kowalski. Predicate logic as programming language. In *IFIP congress*, volume 74, pages 569–544, 1974.

[LB03] Michael Leuschel and Michael Butler. ProB: A model checker for B. In *FME 2003: Formal Methods*, volume 2805, pages 855–874, Berlin, Heidelberg, September 2003. Springer.

[LB05] Michael Leuschel and Michael Butler. Automatic refinement checking for b. In *International Conference on Formal Engineering Methods*, pages 345–359. Springer, 2005.

[LB08] Michael Leuschel and Michael Butler. ProB: An automated analysis toolset for the B method. *International Journal on Software Tools for Technology Transfer*, 10(2):185–203, March 2008.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[Lif99] Vladimir Lifschitz. Answer set planning. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 373–374. Springer, 1999.

[Lif19] Vladimir Lifschitz. *Answer set programming*. Springer Berlin, 2019.

[Llo12] John W Lloyd. *Foundations of logic programming*. Springer Science & Business Media, 2012.

[LM04] Yuliya Lierler and Marco Maratea. Cmodels-2: Sat-based answer set solver enhanced to non-tight programs. In *International Conference on Logic Programming and Non-Monotonic Reasoning*, pages 346–350. Springer, 2004.

[LN09] Claire Lefevre and Pascal Nicolas. The first version of a new asp solver: Asperix. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 522–527. Springer, 2009.

[Lov70] Donald W Loveland. A linear format for resolution. In *Symposium on Automatic Demonstration*, pages 147–162. Springer, 1970.

[Luc70] David Luckham. Refinement theorems in resolution theory. In *Symposium on Automatic Demonstration*, pages 163–190. Springer, 1970.

[LW92] Vladimir Lifschitz and Thomas Y. C. Woo. Answer sets in general nonmonotonic reasoning. pages 603–614. Morgan-Kaufmann, 1992.

[LZ04] Fangzhen Lin and Yuting Zhao. Assat: Computing answer sets of a logic program by sat solvers. *Artificial Intelligence*, 157(1-2):115–137, 2004.

[MD80] Drew McDermott and Jon Doyle. Non-monotonic logic i. *Artificial intelligence*, 13(1-2):41–72, 1980.

[Mey92] Bertrand Meyer. Applying'design by contract'. *Computer*, 25(10):40–51, 1992.

[MG12] Kyle Marple and Gopal Gupta. Galliwasp: A goal-directed answer set solver. In *International Symposium on Logic-Based Program Synthesis and Transformation*, pages 122–136. Springer, 2012.

[MGZ08] Veena S Mellarkod, Michael Gelfond, and Yuanlin Zhang. Integrating answer set programming and constraint logic programming. *Annals of Mathematics and Artificial Intelligence*, 53(1):251–287, 2008.

[Mil93] Harlan D Mills. Zero defect software: Cleanroom engineering. In *Advances in Computers*, volume 36, pages 1–41. Elsevier, 1993.

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):258–282, 1982.

[Moo84] Robert C Moore. Possible-world semantics for autoepistemic logic. Technical report, SRI INTERNATIONAL MENLO PARK CA ARTIFICIAL INTELLIGENCE CENTER, 1984.

[MSCG17] Kyle Marple, Elmer Salazar, Zhuo Chen, and Gopal Gupta. The s(asp) predicate answer set programming system. *The Association for Logic Programming Newsletter*, 2017.

[PL12] Daniel Plagge and Michael Leuschel. Validating b, z and tla+ using prob and kodkod. In *International Symposium on Formal Methods*, pages 372–386. Springer, 2012.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. IEEE, 1977.

[PW76] Michael S Paterson and Mark N Wegman. Linear unification. In *Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 181–186, 1976.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *International Symposium on programming*, pages 337–351. Springer, 1982.

[Rei80] Raymond Reiter. A logic for default reasoning. *Artificial intelligence*, 13(1-2):81–132, 1980.

[Rei81] Raymond Reiter. On closed world data bases. In *Readings in artificial intelligence*, pages 119–140. Elsevier, 1981.

[Rit93] Dennis M Ritchie. The development of the c language. *ACM Sigplan Notices*, 28(3):201–208, 1993.

[Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.

[Rob05] Ken Robinson. A concise summary of the b mathematical toolkit, 2005.

[RT06] Silvio Ranise and Cesare Tinelli. The smt-lib standard: Version 1.2. Technical report, Technical report, Department of Computer Science, The University of Iowa . . . , 2006.

[SBMG07] Luke Simon, Ajay Bansal, Ajay Mallya, and Gopal Gupta. Co-logic programming: Extending logic programming with coinduction. In *International Colloquium on Automata, Languages, and Programming*, pages 472–483. Springer, 2007.

[SMBG06] Luke Simon, Ajay Mallya, Ajay Bansal, and Gopal Gupta. Coinductive logic programming. In *International Conference on Logic Programming*, pages 330–345. Springer, 2006.

[SN01] Tommi Syrjänen and Ilkka Niemelä. The smodels system. In *International Conference on Logic Programming and NonMonotonic Reasoning*, pages 434–438. Springer, 2001.

[Ste] Aix-en-Provence Steria. France. atelier b, user and reference manuals, 1996.

[Syr00] Tommi Syrjänen. Lparse 1.0 user's manual. 2000.

[TJ07] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647. Springer, 2007.

[Tor09] Emina Torlak. *A constraint solver for software engineering: finding models and cores of large relational specifications*. PhD thesis, Massachusetts Institute of Technology, 2009.

[Wei17] Antonius Weinzierl. Blending lazy-grounding and cdnl search for answer-set solving. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 191–204. Springer, 2017.

[Win90] Jeannette M Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8–22, 1990.

[WLBF09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM computing surveys (CSUR)*, 41(4):1–36, 2009.

[WSTL12] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. Swi-prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

[ZS69] Nail'Kalimovich Zamov and Vladislav Ivanovich Sharonov. On a class of strategies for the resolution method. *Zapiski Nauchnykh Seminarov POMI*, 16:54–64, 1969.