Complex Reasoning with Answer Set Programming
Thesis by Publication

Roland Kaminski

Abstract. Answer Set Programming (ASP) allows us to address knowledge-intensive search and optimization problems in a declarative way due to its integrated modeling, grounding, and solving workflow. A problem is modeled using a rule based language and then grounded and solved. Solving results in a set of stable models that correspond to solutions of the modeled problem. In this thesis, we present the design and implementation of the clingo system—perhaps, the most widely used ASP system. It features a rich modeling language originating from the field of knowledge representation and reasoning, efficient grounding algorithms based on database evaluation techniques, and high performance solving algorithms based on Boolean satisfiability (SAT) solving technology. The contributions of this thesis lie in the design of the modeling language, the design and implementation of the grounding algorithms, and the design and implementation of an Application Programmable Interface (API) facilitating the use of ASP in real world applications and the implementation of complex forms of reasoning beyond the traditional ASP workflow.

To succinctly model a wide range of problems, clingo supports a rich input language featuring object variables, function symbols, integer arithmetics, aggregate expressions, and further language constructs to support advanced forms of reasoning. Given a problem modeled in this language, the process of grounding involves replacing all object variables with variable-free terms. Formally, this process results in infinitely many rules and even nested infinite expressions in case of aggregates. Yet, in practice, grounding algorithms yield a finite set of rules consisting of finite subexpressions only that have the same stable models as the semantic representation. We present and prove the correctness of the grounding algorithms employed by the clingo system. After grounding, rules are passed to the solving component. For this purpose, clingo incorporates the clasp solver developed by our research group.

Despite ASP’s versatility, there are problems that are difficult to map to its standard ground and solve workflow. Therefore, we extend the clingo system with a multi-shot solving mode to model flexible reasoning processes and theory reasoning capabilities to handle non-Boolean constraints. Multi-shot solving and theory reasoning are realized via clingo’s API and dedicated language extensions. We show how to apply both approaches to solve temporal problems requiring multiple ground and solve calls, and scheduling problems with constraints over integer variables with large domains.
Chapter 1. Introduction 1
  1.1 Selected contributions 4
  1.2 Overall contributions 8

Chapter 2. Publications 9
  2.1 Potassco: The Potsdam answer set solving collection 9
  2.2 On the foundations of grounding in answer set programming 9
  2.3 Multi-shot ASP solving with clingo 9
  2.4 How to build your own ASP-based system?! 9
  2.5 Temporal answer set programming on finite traces 9
  2.6 Clingo goes linear constraints over reals and integers 10

Chapter 3. Discussion 11
  3.1 The clingo language 11
  3.2 Grounding 16
  3.3 Multi-shot solving 29
  3.4 Temporal solving 42
  3.5 Theory solving 52
  3.6 Conclusion 62
  3.7 Future work 62

Bibliography 64
CHAPTER 1

Introduction

Answer Set Programming (ASP) is a popular approach to solve knowledge-intensive search and optimization problems in a declarative way \[13, 79\]. It rests on strong logic foundations as it closely relates to a non-monotonic variant of the logic of here-and-there \[125\]. Relations to neighboring fields have been intensively studied, for example, ASP can be seen as a fragment of default logic or as an alternative semantics to Prolog that elegantly captures negation as failure \[18, 95\]. These roots make ASP applicable to a wide range of reasoning tasks including tasks involving incomplete information. It features a simple yet powerful rule-based language that can express all problems up to the second level of the polynomial hierarchy. The language allows us to write uniform problem specifications that can be used to solve specific problem instances. Even complex problems can typically be modeled with a small number of generic rules. Another important aspect of ASP is it’s elaboration tolerance. It is often possible to add new rules to a problem specification or modify a few of them to adapt to changing requirements throughout the development of an application. Both problem specification and instance can then be solved by high performance ASP systems. There are numerous applications in various domains that have successfully applied ASP. This includes, for example, systems biology \[108\], planning \[83\], package configuration \[87\], a NASA space shuttle controller \[122\], or scheduling at the Swiss railway company \[1\].

The ASP solving process can be summarized by the steps depicted in Figure 1.0.1. First, a problem is modeled in form of a logic program. The difference to traditional programming is that we do not write an algorithm to solve a problem but specify how solutions to a problem should look like. We then pass this program to an ASP solver. In the following, we use the clingo solver, which is the main subject of this thesis. The

![ASP solving process diagram](image-url)
grounding component of the solver instantiates the logic program by substituting all object variables in it with ground terms. We obtain a ground logic program that is passed to the solving component of Clingo. The solver then reports stable models, each of which represents one solution to the original problem. Notably, the solver is complete—it can find all solutions to a problem.

We explain ASP’s declarative solving approach on the Towers of Hanoi (ToH) puzzle. The goal of this puzzle is to move discs between pegs from an initial to a goal configuration. All discs have different sizes and can be moved between pegs subject to the following conditions: only one disc can be moved at a time, only the top-most disc on a peg can be moved, and a disc cannot be put on a smaller disc. The classical ToH problem has three pegs and \( n \) discs, which initially are on the first peg stacked in order of decreasing size and have to be moved to the third peg. In the following, we develop a logic program that captures solutions to the ToH puzzle for an arbitrary initial and goal configuration up to a fixed predefined number of steps.

First, we represent an instance of the ToH puzzle by a set of facts. We consider a puzzle with three pegs labeled \( a \) to \( c \) and four discs labeled 1 to 4 as depicted in Figure 1.0.2. The four pegs are initially on peg \( a \) and in the goal situation have to be on peg \( c \). The size of a disc is determined by its label; smaller discs are labeled with smaller numbers. The corresponding logic program is given in Listing 1.0.1a together with its grounding in Listing 1.0.1b.

Listing 1.0.1. Facts for ToH instance in Figure 1.0.2

```prolog
#const m=4.
#const n=2**m-1.

time(1..n). peg(a;b;c). disc(1..m).
init(1..m,a).
goal(1..m,c).

(A) Clingo program.

(b) Grounded clingo program.
```

Figure 1.0.2. Towers of Hanoi instance with 4 discs and 3 pegs

The first two lines define the two constants \( m \) and \( n \) where \( m \) is the number of discs set to 4 and \( n \) is the maximum number of moves that can be used to solve the puzzle set to \( 2^m - 1 = 15 \). Both constants are used throughout the program. The remaining lines in the program define facts over the predicates \( \text{time}, \text{peg}, \text{disc}, \text{init}, \text{and goal} \).
We first define facts over predicate \texttt{time} using the constant \texttt{n}. The rule uses the range term \texttt{1..n} as argument of \texttt{time}, which is expanded by the grounding component to the facts \texttt{‘\texttt{time(t).}’} for \texttt{1 \leq t \leq 15}. The purpose of the \texttt{time} predicate is to specify time steps at which discs can be moved. The next line defines the available pegs. In the definition, we use the pool term \texttt{a;b;c}, which is expanded to the facts \texttt{‘\texttt{peg(p).}’} for \texttt{p \in \{a,b,c\}}. The available discs over predicate \texttt{disc} are then defined using a range term as for the time steps. Finally, the \texttt{init} and \texttt{goal} predicates describe the location of the discs at the initial and goal situation. Since we are only interested in configurations where smaller discs are stacked on larger ones, we leave the ordering implicit and only specify which disc is on which peg. Hence, the first argument of predicates \texttt{init} and \texttt{goal} specifies the disc and the second argument the peg. We do not detail the topic here further but this representation is actually advantageous for solving. In fact, in practice, the representation plays an important factor how well a program scales to solve challenging instances.

With the instance at hand, we turn to the specification of the ToH puzzle as a logic program; we also refer to this program as the encoding. The full encoding is given in Listing 1.0.2. In the encoding, we use the predicate \texttt{on} to capture the location of a disc.
1.1 Selected contributions

In the following, we summarize my contributions along the aforementioned model-ground-solve workflow of ASP and give abstracts of the papers included in this thesis. We begin with an overview article of the systems I have contributed to (see Section 1.1.1) and
$\text{clingo instance.lp encoding.lp 0}
\text{clingo version 5.5.2}
\text{Reading from instance.lp ...}
\text{Solving...}
\text{Answer: 1}
\text{move(1,a,b,1) move(2,a,c,2) move(3,a,b,4) move(1,b,c,3)}
\text{move(1,c,a,5) move(2,c,b,6) move(1,a,b,7) move(4,a,c,8)}
\text{move(1,b,c,9) move(2,b,a,10) move(1,c,a,11) move(3,b,c,12)}
\text{move(1,a,b,13) move(2,a,c,14) move(1,b,c,15)}
\text{SATISFIABLE}
\text{Models : 1}
\text{Calls : 1}
\text{Time : 0.02s (Solving: 0.01s 1st Model: 0.01s Unsat: 0.00s)}
\text{CPU Time: 0.02s}

\text{Listing 1.0.3. Clingo output for example ToH problem}

then turn to two foundational articles regarding grounding (see Section 1.1.2) and multi-shot solving (see Section 1.1.3). The third paper introduces theory solving, providing the means to extend ASP with foreign inferences (see Section 1.1.4). The final two papers show successful ASP-based systems. We present systems to extend ASP with integer constraints (see Section 1.1.6) using theory solving and temporal operators (see Section 1.1.5) relying on multi-shot solving.

\text{1.1.1 Potassco: the Potsdam answer set solving collection.}\ The paper provides an overview over the core systems for ASP solving developed within the Potassco project at the University of Potsdam. The two central components of the project are the grounder \textit{gringo} and the (propositional) solver \textit{clasp}. Relying on these systems, \textit{clingo}, \textit{iclingo}, \textit{claspD}, and \textit{claspar} offer extended functionality. The \textit{clingo} system is the monolithic combination of \textit{clasp} and \textit{gringo} providing the user with the convenience of not having to deal with two different programs. The \textit{iclingo} system extends \textit{clingo} with an incremental grounding and solving mode, which can for example be used to solve planning problems. The \textit{claspD} solver adds support for disjunctive logic programs to \textit{clasp} allowing the user to solve problems on the second level of the polynomial hierarchy. Finally, there is the \textit{claspar} system, which was the first ASP system to enable parallel CDNL-based ASP solving on clusters of computers.

We published the paper in 2011. It provides an overview over the systems developed in our group and with more than 500 citations is one of our most cited papers. My contributions to the paper lie in the design and implementation of the \textit{gringo}, \textit{clingo}, \textit{iclingo}, and \textit{claspar} systems as well as the design of the input languages of both \textit{gringo} and \textit{iclingo}. The \textit{gringo} and \textit{clasp} systems provide the foundations for many advanced systems developed by our group [12, 22, 100, 102, 104] and others [9, 11, 26, 52, 53, 127, 130, 138, 143]. The \textit{claspar} system can be seen as a predecessor to the multi-threaded \textit{clasp} system, which offers similar functionality for the nowadays ubiquitous multi-core architecture. The functionalities of \textit{clasp}, \textit{gringo}, and \textit{iclingo} have meanwhile been
1.1. SELECTED CONTRIBUTIONS

1.1.2 On the foundations of grounding in answer set programming. The paper provides the theoretical foundations of grounding algorithms. Building on the semantics of gringo’s modeling language defined in [69] and the operator based characterization of stable and well-founded models in [139], we introduce a formal characterization of grounding algorithms in terms of (fixed point) operators. A major role is played by dedicated well-founded operators whose associated models provide semantic guidance for delineating the result of grounding along with on-the-fly simplifications. We address an expressive class of logic programs that incorporates recursive aggregates and thus amounts to the scope of existing ASP modeling languages. This is accompanied with a plain algorithmic framework detailing the grounding of recursive aggregates. The given algorithms correspond essentially to the ones used in the ASP grounder gringo.

We published this paper in 2022. Here, I provide a detailed description of the grounding algorithms used in the gringo system. While semi-naive based grounding algorithms were first implemented in dlv, this is the first paper that provides a tight characterization of the output of such an (optimized) algorithm and a full proof for its soundness. Notably, we rely on advanced concepts like infinitary formulas to capture aggregate expressions. We include the paper in Section 2.2 and discuss the grounding algorithms in Section 3.2.

1.1.3 Multi-shot ASP solving with clingo. This paper introduces a flexible paradigm of grounding and solving in ASP, which we refer to as multi-shot ASP solving, and present its implementation in the ASP system clingo (version 4). Multi-shot ASP solving features grounding and solving processes that deal with continuously changing logic programs. In doing so, they remain operative and accommodate changes in a seamless way. For instance, such processes allow for advanced forms of search, as in optimization or theory solving, or interaction with an environment, as in robotics or query answering. Common to them is that the problem specification evolves during the reasoning process, either because data or constraints are added, deleted, or replaced. This evolutionary aspect adds another dimension to ASP since it brings about state changing operations. We address this issue by providing an operational semantics that characterizes grounding and solving processes in multi-shot ASP solving. This characterization provides a semantic account of grounder and solver states along with the operations manipulating them. The operative nature of multi-shot solving avoids redundancies in relaunching grounder and solver programs and benefits from the solver’s learning capacities. Clingo accomplishes this by complementing ASP’s declarative input language with control capacities. On the declarative side, a new directive allows for structuring logic programs into named and parametrizable subprograms. The grounding and integration of these subprograms into the solving process is completely modular and fully controllable from the procedural side. To this end, clingo offers a new Application Programming Interface (API) that is conveniently accessible via external languages. By strictly separating logic and control, clingo also abolishes the need for dedicated systems for incremental and
reactive reasoning, like iclingo and oclingo, respectively, and its flexibility goes well beyond the advanced yet still rigid solving processes of the latter.

We published this paper in 2019. My contributions are the design and implementation of the application programmable interface (API) of the clingo system and its experimental evaluation. The clingo API comprises more than 200 functions and is available for C, C++, Lua and Python. The development includes both the grounding and solving aspect of ASP. While the grounder is solely implemented by me, the solving functionality relies on the infrastructure provided by the clasp system. We include the paper in Section 2.3 and discuss multi-shot solving in Section 3.3.

1.1.4 How to build your own ASP-based system. This paper provides a tutorial on how to use clingo (version 5) to build customized ASP-based systems. More precisely, we show how the ASP system clingo can be used for extending ASP and for implementing customized special-purpose systems. To this end, we propose two alternatives. We begin with a traditional AI technique and show how metaprogramming can be used for extending ASP. This is a rather light approach that relies on clingo’s reification feature to use ASP itself for expressing new functionalities. The second part of this tutorial uses traditional programming (in Python) for manipulating clingo via its API. This approach allows for changing and controlling the entire model-ground-solve workflow of ASP. Central to this is clingo’s new Application class that allows us to draw on clingo’s infrastructure by customizing processes similar to the one in clingo. For instance, we may apply manipulations to programs’ abstract syntax trees, control various forms of multi-shot solving, and set up theory propagators for foreign inferences. A cross-sectional structure, spanning meta as well as application programming, is clingo’s intermediate format, aspif, that specifies the interface among the underlying grounder and solver. We illustrate the aforementioned concepts and techniques throughout this tutorial by means of examples and several non-trivial case-studies. In particular, we show how clingo can be extended by difference constraints and how guess-and-check programming can be implemented with both meta and application programming.

We published this paper in 2020. My contributions lie in both providing the infrastructure to build applications as well as building applications on top of clingo. This includes the development of the reification output and basic metaencodings as well as the interfaces for multi-shot and theory solving. I contributed to all applications relying on the clingo API presented in the paper. This includes extending ASP with difference constraints and solving second level problems using a guess and check approach. We include the paper in Section 2.4 and discuss the paper in Sections 3.4 and 3.5.

1.1.5 Temporal answer set programming on finite traces. In this paper, we introduce an alternative approach to Temporal Answer Set Programming that relies on a variation of Temporal Equilibrium Logic (TEL) for finite traces. This approach allows us to even out the expressiveness of TEL over infinite traces with the computational capacity of multi-shot ASP solving. Also, we argue that finite traces are more natural when reasoning about action and change. As a result, our approach is readily implementable via multi-shot ASP systems and benefits from an extension of ASP’s full-fledged input language with temporal operators. This includes future as well as past operators whose combination offers a rich temporal modeling language. For computation, we identify the
class of temporal logic programs and prove that it constitutes a normal form for our approach. Finally, we outline two implementations, a generic one and an extension of the ASP system clingo.

We published this paper in 2018. My contributions lie in the design and implementation of the two systems tel and telingo developed in this paper as well as work on the normal form for temporal programs. Notably, the telingo system gave rise to the design and implementation of the API to modify the abstract syntax tree of clingo’s input. We include the paper in Section 2.5 and discuss the basic telingo implementation in Section 3.4.

1.1.6 Clingo goes linear constraints over reals and integers. While we can use ASP to solve a wide range of knowledge-intensive combinatorial search problems, it falls short in handling non-Boolean constraints like linear constraints over integers. Such kind of constraints play an important role in solving industrial problems like, for example, train scheduling at the Swiss railway company [1]. In the paper, we instantiate clingo’s theory reasoning framework with different forms of linear constraints and elaborate upon its formal properties. We discuss three different implementations, and present techniques for using these constraints in a reactive context. More precisely, we introduce extensions to clingo with difference and linear constraints over integers and reals, respectively, and realize them in complementary ways. Finally, we empirically evaluate the resulting clingo derivatives clingcon, clingo[dl], and clingo[lp] on common language fragments and contrast them to related ASP systems.

We published this paper in 2017. My contributions lie in the design and implementation of the API to create the systems based on clingo. In particular, the applications in this paper drove the development of its theory propagation related functionality. Furthermore, I refined both the Python and C++ implementation of clingo[dl]. Notably, the C++ variant of clingo[dl] performed best in the benchmarks presented in this paper. We include the paper in Section 2.6 and discuss the basic clingo[dl] implementation in Section 3.5.

1.2 Overall contributions

I contributed to the following papers loosely grouped by topics: the ASP input language [31,69], grounding [84,86,89], solving [56,77,78,82], multi-shot solving [67,68,73,75,80,83,109], theory solving [27–29,61,76,104], portfolio solving [81,100,101], systems biology [108,126], package configuration [87,129], advanced modeling [70,71,85,88], and overview articles [72,74].

Furthermore, clingo is one of the most widely used systems for ASP solving. This can be seen in the number of researchers citing our work. For example, clingo is used in many systems [9,11,26,52,54,127,130,138] and applications [4,16,20,21,38,42,49,58,59,66,103,118,120,131,133,145] by third parties worldwide.
CHAPTER 2

Publications

This chapter lists the publications discussed in this thesis.

2.1 *Potassco: The Potsdam answer set solving collection*

**Authors:** Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, 
Torsten Schaub and Marius Schneider

**Published in:** AI Communications 24 (2011) 107–124

**Bibliography entry:** [74]

**DOI:** 10.3233/AIC-2011-0491

2.2 *On the foundations of grounding in answer set programming*

**Authors:** Roland Kaminski and Torsten Schaub

**Published in:** Theory and Practice of Logic Programming (2022) 1–60

**Bibliography entry:** [107]

**DOI:** 10.1017/S1471068422000308

2.3 *Multi-shot ASP solving with clingo*

**Authors:** Martin Gebser, Roland Kaminski, Benjamin Kaufman and Torsten Schaub

**Published in:** Theory and Practice of Logic Programming 19 (2019) 27–82

**Bibliography entry:** [80]

**DOI:** 10.1017/S1471068418000054

2.4 *How to build your own ASP-based system?*

**Authors:** Roland Kaminski, Javier Romero, Torsten Schaub, and Philipp Wanko

**Published in:** Theory and Practice of Logic Programming 23 (2023) 299–361

**Bibliography entry:** [106]

**DOI:** 10.1017/S1471068421000508

2.5 *Temporal answer set programming on finite traces*

**Authors:** Pedro Cabalar, Roland Kaminski, Torsten Schaub and Anna Schuhmann

**Published in:** Theory and Practice of Logic Programming 18 (2018) 406–420

**Bibliography entry:** [29]

**DOI:** 10.1017/S1471068418000297
2.6 *Clingo* goes linear constraints over reals and integers

**Authors:** Tomi Janhunen, Roland Kaminski, Max Ostrowski, Torsten Schaub, Sebastian Schellhorn, and Philipp Wanko

**Published in:** Theory and Practice of Logic Programming 17 (2017) 872–888

**Bibliography entry:** [104]

**DOI:** 10.1017/S1471068417000242
CHAPTER 3

Discussion

3.1 The clingo language

One of the most important aspects of ASP is its high-level input language that allows for modeling combinatorial problems in an easy and succinct way. Historically, there are the input languages of the lparse [135] and dlv-1 [113] systems. The lparse system was the first ASP grounder followed by the grounder of the dlv-1 system.1 Below we list the main features and differences of the two systems focussing on their input languages. Both systems support rules build from atoms over object variables and constants but differ regarding extended language features and accepted input. The lparse input language supports both symbolic and interpreted functions providing a natural way to model many problems. The dlv-1 system does not support symbolic functions and instead of providing interpreted functions, requires the user to use inbuilt predicates that only support non-negative integers and additionally require a maximum value for their domain. Furthermore, both lparse and dlv-1 require variables in rules to be bound by some positive body literal. In dlv-1, any positive body literal can bind a variable. In lparse, only omega-restricted [136] positive body literals can bind variables restricting the class of programs accepted by lparse as compared to dlv-1. Another important language element of ASP are aggregate expressions to conveniently and compactly model properties involving sets of atoms. The lparse system supports choice rules, cardinality constraints, and weight constraints. Recursion through aggregates is supported even though there are counterintuitive cases for non-monotone aggregates [64]. The dlv-1 system supports #count, #sum, and #min and #max aggregates. The #count and #sum aggregates correspond roughly to cardinality and weight constraints. Unlike lparse, dlv-1 only supports stratified aggregates.

Section 2.1 comprises the work in [74] introducing the open source project potassco, the Potsdam Answer Set Solving Collection bundling tools for ASP developed at the University of Potsdam. These tools include the grounder gringo and solver clasp that are nowadays combined in the ASP system clingo providing many advanced features like multi-shot or theory solving, which we discuss in Sections 3.3 to 3.5. For this reason, we only consider the clingo language in this section, which some of our earlier publications also refer to as gringo language. We begin by briefly describing the evolution of the language from clingo series 3 to 5. In [74], we give an overview of the main features of the clingo-3 input language. This version of the language combines the main features of the lparse and dlv-1 input languages. Its syntax for aggregates follows that of lparse and is incompatible with dlv-1. In version 4 of the language, we aligned the aggregate

---

1Meanwhile the dlv-2 system supersedes dlv-1 featuring an extended input language.
syntax of clingo with that of dlv-1. Starting with clingo-4 and dlv-2, both systems accept
a common syntax as defined in the ASP-Core-2 standard that has been used in ASP
competitions \cite{5,32,92,93} to evaluate the performance of ASP solvers. However, the
clingo-4 language is a superset of the ASP-Core-2 standard. In \cite{69}, we refine and
formally define the semantics of the clingo-4 language. The refinements concern features
beyond ASP-Core-2, like the semantics of pools, undefined arithmetic, and recursive
aggregates, and are implemented in the clingo-5 system.

3.1.1 Background. In the following, we do not consider the full clingo-5 input
language. Instead, we introduce a rather simple class of logic programs and define its
semantics. Note, however, that this class is powerful enough to model NP-complete
problems \cite{119}. We use this class of programs throughout this chapter to show important
aspects of the clingo system.

We consider a signature $\Sigma = (\mathcal{C}, \mathcal{P}, \mathcal{V})$ of finite disjoint sets of constant, predicate,
and variable symbols. Predicate symbols are associated with non-negative integer arities.
In the following, we use a finite subset of the non-negative integers as constant symbols,
lower case strings for predicate symbols, and upper case strings for variable symbols. We
also drop the term ‘symbol’ and speak of constants, predicates, and variables.

An atom over signature $\Sigma$ has form $p(t_1, \ldots, t_n)$ where $p \in \mathcal{P}$ is a predicate with arity
$n$ and each $t_i \in \mathcal{C} \cup \mathcal{V}$ is either a constant or variable. Given an atom $a$ over signature $\Sigma$,
a literal over $\Sigma$ is either the atom itself or its negation $\neg a$. A literal without negation is
called positive, and negative otherwise.

A rule over signature $\Sigma$ has form $h \leftarrow B$ where $h$ is an atom over $\Sigma$ and $B$ is a finite
set of literals over $\Sigma$. We refer to $h$ as the head, $B$ the body, and the literals in $B$ as
the body literals of $r$. A program over signature $\Sigma$ is a finite set of rules over $\Sigma$. In the
following, we omit braces around sets and write $h \leftarrow l_1, \ldots, l_n$ instead of $h \leftarrow \{l_1, \ldots, l_n\}$
when explicitly writing the literals $l_i$, and only write the rule head when the body is empty.

We say that an atom, literal, rule, or program is ground if it does not contain
any variables. We use $H(h \leftarrow B) = h$ and $B(h \leftarrow B) = B$ to obtain the head
and body of a rule, respectively. We extend both functions to programs by letting
$H(P) = \{H(r) \mid r \in P\}$ and $B(P) = \bigcup_{r \in P} B(r)$. Given a set of literals $L$, the
sets $L^+ = \{l \in L \mid l$ is an atom$\}$ and $L^- = \{a \mid \neg a \in L\}$ comprise all atoms occurring
positively and negatively in $L$, respectively; to refer to all atoms in $L$, we use $L^\pm = L^+ \cup L^-$. Furthermore, we let $A(P) = H(P) \cup B(P)^\pm$ stand for the set of all atoms occurring
in a program $P$. Finally, we use $\text{pred}(p(t_1, \ldots, t_n)) = p$ to refer to the predicate of
atom $p(t_1, \ldots, t_n)$ and $\text{pred}(A) = \{\text{pred}(a) \mid a \in A\}$ to refer to the predicates occurring
in a set $A$ of atoms.

Example 3.1.1. As an example, we consider a simplified version of the introductory
case in Listings 1.0.1 and 1.0.2. Since we use this example throughout the following
sections, we abbreviate predicates to keep listings and figures compact. We use predicates
e (equal), ne (notequal), o (on), no (noton), m (move), nm (notmove), and f (fail).
For our instance and encoding, we use a signature with the symbols

- $\mathcal{P} = \{\text{step}, \text{peg}, \text{disc}, \text{init}, e, ne, o, no, m, nm, f\}$,
- $\mathcal{V} = \{P, P', D, T, T'\}$, and
3.1. THE clingo LANGUAGE

step(0, 1)  step(1, 2)
peg(1)      peg(2)
disc(1)     disc(2)
init(1, 1)  init(2, 1)

Listing 3.1.1. Simplified ToH instance

\[ (r_1) \quad e(P, P) \leftarrow peg(P) \]
\[ (r_2) \quad ne(P', P) \leftarrow peg(P'), peg(P), \neg e(P', P) \]
\[ (r_3) \quad o(D, P, 0) \leftarrow init(D, P) \]
\[ (r_4) \quad m(D, P, T) \leftarrow \neg nm(D, P, T), o(D, P', T'), ne(P', P), step(T', T) \]
\[ (r_5) \quad nm(D, P, T) \leftarrow \neg m(D, P, T), o(D, P', T'), ne(P', P), step(T', T) \]
\[ (r_6) \quad o(D, P, T) \leftarrow m(D, P, T) \]
\[ (r_7) \quad o(D, P, T) \leftarrow o(D, P, T'), \neg no(D, P, T), step(T', T) \]
\[ (r_8) \quad no(D, P, T) \leftarrow o(D, P', T), ne(P', P) \]
\[ (r_9) \quad f \leftarrow \neg f, o(D, P, T), no(D, P, T) \]

Listing 3.1.2. Simplified ToH encoding

- \( \mathcal{C} = \{0, 1, 2\} \).

The problem instance is given in Listing 3.1.1. Since our simple language neither provides arithmetics nor comparisons, we use predicate `step` instead of `time` to capture transitions from one time point to the next. We also ignore the goal just focusing on transitions.

The encoding in Listing 3.1.2 encodes moves of discs between pegs but neither asserts that only the smallest disc on a peg is moved nor the goal condition is reached; in fact, an arbitrary number of discs can be moved at each but the initial time step. We only highlight the differences as compared to the encoding in Listing 1.0.2 in the introductory section.

In our simple language, we neither have equality nor non-equality predicates, which we encode using auxiliary predicates and rules. Moreover, we neither have choice rules nor integrity constraints at our disposal, which, instead, we encode using even and odd cycles \[116\], respectively. Finally, we only have default negation (via the \( \neg \) connective) at our disposal. We encode classical negation using auxiliary predicates for classically negated atoms \[96\], and odd cycles to discard contradictory models.

We begin by encoding when two pegs are equal or different in rules \( r_1 \) and \( r_2 \), respectively. Moreover, since there are no aggregates in our language, we keep the encoding short by allowing arbitrary moves in a transition. To determine candidate moves, we use rules \( r_4 \) and \( r_5 \), which depend cyclically on each other via an even cycle involving predicates \( m \) and \( nm \). This cycle ensures that a disc is either moved or not moved.
Finally, we use an odd cycle involving atom \( f \) in rule \( r_9 \) to discard inconsistent solutions regarding the locations of discs. This is important for instances with more than two pegs because there is no constraint that a disc can only be moved to one peg. Here, we emulate classical negation because corresponding atoms over \( o \) and \( no \) can never be part of a model together.

An instance of a rule is obtained by substituting constants from its signature for all its variables. We use \( \Gamma(P) \) to denote the set of all instances of rules in \( P \).

**Example 3.1.2.** For example, the set of all instances of rule \( r_1 \) is

\[
\begin{align*}
(\ell_1) & \quad e(0, 0) \leftarrow \text{peg}(0) \\
(\ell_2) & \quad e(1, 1) \leftarrow \text{peg}(1) \\
(\ell_3) & \quad e(2, 2) \leftarrow \text{peg}(2)
\end{align*}
\]

because we have \( C = \{0, 1, 2\} \). Observe that for a rule with \( n \) variables, we obtain \(|C|^n\) instances. Since rule \( r_1 \) has one variable and we have three constants, we obtain three instances as expected. We also see that the first instance refers to atom \( \text{peg}(0) \), which can never be derived by any rule. In Section 3.2, we present algorithms to compute instances of rules avoiding such unnecessary ones.

The atom base of a signature \( \Sigma \) is the set of all ground atoms over \( \Sigma \). A two-valued interpretation over an interpretation \( \Sigma \) is a set \( I \subseteq A \) of atoms where \( A \) is the atom base of \( \Sigma \). Atoms contained in \( I \) are considered true and atoms contained in \( A \setminus I \) are considered false.

From now on we no longer mention the signature and assume that interpretations, programs, rules, literals, and atoms in the same context share the same signature. An interpretation \( I \) satisfies

- an atom if it is true in \( I \),
- a literal \( \neg a \) if \( a \) is false in \( I \),
- a rule if \( I \) satisfies its head or does not satisfy some literal in its body, and
- a program if \( I \) satisfies all its rules.

If \( I \) satisfies a program, we say that \( I \) is a model of the program.

The reduct \( P^I \) of a program \( P \) w.r.t. to an interpretation \( I \) is the program \( \{ H(r) \leftarrow B(r)^+ \mid r \in \Gamma(P), I \cap B(r)^- = \emptyset \} \), which selects rule instances from \( P \) such that all negative body literals are satisfied by the interpretation and strips negative body literals \([95]\). An interpretation \( I \) is a stable model of a program \( P \) if it is among the subset minimal models of the reduct \( P^I \). Observe that a stable model \( I \) is also a model because the reduct only removes rules satisfied by \( I \) and if a reduced rule is satisfied by \( I \), so is its unreduced counterpart.

**Example 3.1.3.** For example, let us consider the program \( P \) consisting of the rules in Listings 3.1.1 and 3.1.2. Below, we specify the stable models of our program. To keep the example compact, we do not show that the models are indeed subset minimal models of the reduct.

Let \( F \) be the heads of the rules in Listing 3.1.1. Any stable model of \( P \) includes the atoms from set \( F \). Further atoms are determined by the rules \( r_1 \) to \( r_9 \). In particular, rules \( r_4 \) and \( r_5 \) specify that a disc is either moved or not. Because we can move any
disc at any time, the set $M = \{1,2\} \times \{1,2\}$ captures all possible moves of discs at time steps. Let $M \subseteq \mathcal{M}$ be a set of moves and $M = \mathcal{M} \setminus M$ be its complement. Because there are only two pegs, the set $M$ uniquely determines the stable model $X_0 \cup X_1 \cup X_2$ of our program where the sets $X_i$ of atoms relevant to time steps $0 \leq i \leq 2$ are

\[
X_0 = F \\
\cup \{ e(p, p) \mid p \in \mathcal{C}, \text{peg}(p) \in F \} \\
\cup \{ ne(p, p') \mid p, p' \in \mathcal{C}, \text{peg}(p'), \text{peg}(p) \in F, p \neq p' \} \\
\cup \{ o(d, p, 0) \mid d, p \in \mathcal{C}, \text{init}(d, p) \in F \} \\
\cup \{ no(d, p, 0) \mid d, p \in \mathcal{C}, \text{init}(d, p) \in F, \text{peg}(p) \in F, p \neq p' \}, \text{ and} \\
X_i = \{ m(d, p, i), o(d, p, i) \mid (d, i) \in M, p \in \mathcal{C}, \text{no}(d, p, i - 1) \in X_{i - 1} \} \\
\cup \{ nm(d, p, i), \text{no}(d, p, i) \mid (d, i) \in \mathcal{M}, p \in \mathcal{C}, \text{no}(d, p, i - 1) \in X_{i - 1} \} \\
\cup \{ o(d, p, i) \mid (d, i) \in M, p \in \mathcal{C}, o(d, p, i - 1) \in X_{i - 1} \} \\
\cup \{ \text{no}(d, p, i) \mid (d, i) \in M, p \in \mathcal{C}, o(d, p, i - 1) \in X_{i - 1} \}.
\]

Note that the atoms in $X_0$ are uniquely determined by the facts.

3.1.2 Summary. Section 2.1 comprises [74] presenting an overview of the systems developed within the Potassco project. This overview includes a short summary of the clingo-3 language that at that point was already widely used by researchers. The clingo-3 language supports safe disjunctive programs with aggregates and is backward compatible with the lparse language regarding the main language features. This includes choice rules, cardinality and weight constraints, conditional literals, and term pools. Notably, clingo-3 already has extended support for aggregates using a syntax similar to weight constraints. This includes $\#\text{min}$, $\#\text{max}$, $\#\text{count}$, $\#\text{sum}$, $\#\text{avg}$, and $\#\text{times}$ aggregates.

The language was modified in clingo series 4 to be more compatible with dlv’s input language. The common part of the input languages are now specified in the ASP-Core [34] and ASP-Core-2 [31] standards as supported by clingo-4 and dlv-2. At this point, we also dropped backward compatibility with lparse. One major change is the removal of support for lparse’s multi-set based weight constraints of form ‘[ ... ] b’ in favor of set based aggregates of form ‘$\#\text{sum} \{ ... \} \leq b$’. However, we retained and extended support for some language constructs introduced in lparse. This includes choice rules, cardinality constraints as a shortcut for $\#\text{count}$ aggregates, conditional literals, and term pools. Support for less frequently used aggregates like $\#\text{avg}$ and $\#\text{times}$ has been dropped.

The semantics of clingo series 5 was further cleaned up and refined in [69] regarding the semantics of term pools, undefined arithmetic, and aggregates. Term pools were introduced in lparse and provide a convenient way to compactly express a set of rules. We refined their semantics and ensured that they can be consistently used throughout a logic program. Undefined arithmetic was simply replaced by value 0, which could lead to surprising solutions. It is now treated like an empty pool; in effect, discarding rules or nested language constructs involving undefined operations. Using the results in [6], the system now also supports recursive non-monotone aggregates.
Apart from this, the language supports heuristic directives to incorporate domain-specific information into ASP solving [90], script, external and program directives to support multi-shot solving (see Section 3.3), and theory definitions and atoms to incorporate theory specific reasoning into ASP (see Sections 3.4 and 3.5).

3.2 Grounding

ASP rests upon a two step approach consisting of grounding and solving. Interestingly, there are a lot of systems only accepting ground programs as input, facing few systems accepting non-ground programs, for example, lparse (which only implements grounding), dlv, or clingo. This is probably due to the complexity of the input language as compared to the much simpler ground format. One such complex and important feature of the input language are aggregate expressions. This can be seen in the numerous works [47, 62, 65, 98, 110, 111, 134] on finding a suitable semantics for aggregates. Among them, we follow the semantics introduced in [65] defined for the ground case.

In the non-ground case, we are faced with one further challenge. Grounding involves replacing variables with all possible variable-free terms constructible from the signature associated with a program. Since programs can contain function symbols and (symbols for) integers, the systematic grounding of a program has infinite size as soon as there is one variable. The semantics in [65] covers programs with infinitely many rules. But to capture aggregates, we also have to deal with nested expressions composed of infinitely many elements. The semantic foundations for aggregates (and the main language constructs of the clingo language) have been laid in [69]. It resorts to infinitary formulas that support nested conjunctions and disjunctions of infinite size.

Section 2.2 comprises the work in [107], which presents a grounding algorithm that takes programs with aggregates as input. We show that the output of the algorithm can be characterized by fixed points of operators. In particular, we use an approximation of the well-founded model of a program [142], that is, the fixed point of the well-founded operator. We build a ground program that is equivalent to the input program while computing an approximate model. Deciding whether the approximate model is finite is undecidable in general but there are classes of programs for which the fixed points of our operators are finite and, thus, also our algorithm terminates in finitely many steps. In what follows, we use the rather simple class of programs introduced in Section 3.1 and develop grounding algorithms for it. These algorithms can be seen as a baseline for the algorithms in Section 2.2. While it is relatively straightforward to extend the algorithms to accept programs with aggregates, it is much harder to characterize them and formally prove their correctness. Thus, we refer for detailed results to Section 2.2 and only discuss some issues regarding aggregate programs and our overall contributions at the end of the section.

3.2.1 Background. In Section 3.1, we introduced a reduct based characterization of the stable models of programs. For the purpose of grounding, we now turn to an operator based characterization of their semantics following [140].

As put forward in [141], we associate a program $P$ with its one-step provability operator $T_P$, defined for an interpretation $I$ as $T_P(I) = \{H(r) \mid r \in \Gamma(P), B(r)^+ \subseteq I, B(r)^- \cap I = \emptyset\}$, which gathers all rule heads whose body literals are satisfied by the
interpretation. The fixed points of the operator $T_P$ w.r.t. $\subseteq$ are the supported models and the prefixed points w.r.t. $\subseteq$ the models of the program $P$. A model either satisfies the head or falsifies the body of all rule instances of the program; a supported model \textsuperscript{37} additionally requires that for each atom in the supported model there is at least one rule instance with a satisfied body supporting it.

**Example 3.2.1.** For example, just considering the three rule instances $g_1$ to $g_3$ of rule $r_1$ from Example 3.1.2, the empty set is a model and the only supported model of the three rule instances, whereas the set $\{e(i, i), p(i) \mid 0 \leq i \leq 2\}$ is a model but not a supported model.

The next operator uses the reduct from Section 3.1. We observe that the reduct of a program is positive, that is, it does not contain rules with negative body literals. It turns out that the $T_P$ operator always has a least fixed point for positive programs w.r.t. the $\subseteq$ relation, which we write as $\text{lfp}_\subseteq(T_P)$. Given a program $P$, we define the stable operator $S_P$ applied to an interpretation $I$ as

$$S_P(I) = \text{lfp}_\subseteq(T_P I).$$

The fixed points of $S_P$ w.r.t. $\subseteq$ correspond to the stable models of $P$ as defined in Section 3.1. The operator based characterization of stable models emphasizes that all atoms in a stable model are acyclically derivable. Fixing the negative literals via the reduct, all atoms in the stable model must be derivable via the $T$ operator. Note that the program in Example 3.1.3 is a so-called tight program \textsuperscript{63} for which the supported and stable models coincide.

We next turn to operators that take additional truth values into account. A four-valued interpretation over signature $\Sigma$ is a pair $(I, J)$ of sets $I \subseteq A$ and $J \subseteq A$ of atoms where $A$ is the atom base of $\Sigma$. Atoms contained in $I$ are considered certain and atoms in $J$ possible. Intuitively, an atom that is certain and possible is true, an atom that is certain but not possible is inconsistent, an atom that is not certain but possible is unknown, and an atom that is neither certain nor possible is false.

Let $(I', J')$ and $(I, J)$ be two four-valued interpretations over the same signature. We use $(I, J) \cup (I', J')$ as a shortcut for $(I \cup I', J \cup J')$. Furthermore, we say that $(I', J')$ is less precise than $(I, J)$, written $(I', J') \leq_p (I, J)$, if $I' \subseteq I$ and $J \subseteq J'$. The precision ordering also has an intuitive reading: the more atoms are certain or the fewer atoms are possible, the more precise is a four-valued interpretation. The least precise four-valued interpretation over signature $\Sigma$ is $(\emptyset, A)$ where $A$ is the atom base of $\Sigma$.

We omit whether an interpretation is two- or four-valued whenever clear from context.

As put forward in \textsuperscript{142}, the stable operator has an alternating fixed point that can be used to define the well-founded model of a program. We follow the slightly different approach of \textsuperscript{46}, by constructing an operator that takes a four-valued interpretation as input. Given a program $P$, the well-founded operator $W_P$ for an interpretation $(I, J)$ is defined as

$$W_P(I, J) = (S_P(J), S_P(I)).$$
This operator has a least fixed point w.r.t. \( \leq_p \), which we refer to as the **well-founded model** of \( P \) and write as

\[
WM(P) = \text{lfp}_{\leq_p}(WP).
\]

Given a program \( P \) and its well-founded model \((I, J)\), we have \( I \subseteq X \subseteq J \) for any stable model \( X \) of \( P \).

**Example 3.2.2.** Coming back to the program from Example 3.1.1, we give its well-founded model \((I, J)\). The certain atoms \( I \) correspond to the set \( X_0 \) (derived from facts independent of \( M \)) given in Example 3.1.3. This agrees with the intuition that all atoms derivable from the certain atoms given by the instance should be certain in the well-founded model as well. We now turn to the possible atoms \( J = J_0 \cup J_1 \cup J_2 \) where the sets \( J_i \) of possible atoms relevant to time steps \( 0 \leq i \leq 2 \) are

\[
J_0 = I, \quad J_i = \{ m(d, p, i), nm(d, p, i), o(d, p, i), no(d, p, i) \mid d, p \in C, no(d, p, i-1) \in J_{i-1} \} \\
\cup \{ o(d, p, i), no(d, p, i) \mid d, p \in C, o(d, p, i-1) \in J_{i-1} \} \\
\cup \{ f \}.
\]

Note that at time step one, only moves to peg two are considered. At time step two, moves to all pegs are possible. Furthermore, observe that atom \( f \) is possible in the well-founded model but never part of a stable model; all other atoms possible in the well-founded model are contained in at least one stable model.

Proceeding as in Section 2.2, the **simplification** \( P^{I,J} \) of a program \( P \) w.r.t. the interpretation \((I, J)\) is \( \{ r \in \Gamma(P) \mid B(r)^+ \subseteq J, B(r)^- \cap I = \emptyset \} \). The program \( P \) and its simplification \( P^{I,J} \) have the same stable models for all \((I, J) \leq_p WM(P)\). In practice, grounding algorithms step-by-step calculate the rules contained in the simplification relative to some approximation of the well-founded model.

**Example 3.2.3.** For example, the rule instance \( g_1 \) is discarded if our example program is simplified with its well-founded model \((I, J)\) because \( p(0) \notin J \). Rule instances \( g_2 \) and \( g_3 \) are kept.

One of the first steps during grounding is to group rules into components suitable for successive instantiation. This amounts to splitting a program into a sequence of subprograms.

Inter-rule dependencies are determined by the predicates appearing in their heads and bodies. A rule \( r_1 \) depends on a rule \( r_2 \) if \( \text{pred}(H(r_1)) \in \text{pred}(B(r_2))^\pm \). Rule \( r_1 \) depends positively or negatively on \( r_2 \) if \( \text{pred}(H(r_1)) \in \text{pred}(B(r_2))^+ \) or \( \text{pred}(H(r_1)) \in \text{pred}(B(r_2))^− \), respectively. The strongly connected components of a program \( P \) are the equivalence classes of the rule dependency relation. A strongly connected component \( P_1 \) depends on another component \( P_2 \) if there is a rule in \( P_1 \) that depends on a rule in \( P_2 \). A topological ordering of the components is then used to guide grounding. We define an **instantiation sequence** for a program \( P \) as the sequence \((P_i)_{i \leq 1}\) of its strongly connected components such that \( i < j \) if \( P_j \) depends on \( P_i \).
3.2. GROUNDING

Example 3.2.4. Figure 3.2.1 depicts the inter-rule dependencies of our example program in Listings 3.1.1 and 3.1.2. Rules are depicted in solid boxes and their dependencies via arrows, which join/branch in case that there is a small solid circle. A rule $r$ depends on another rule $r'$ if we can follow from the box of $r$ beginning without an arrow tip to the one of $r'$ with an arrow tip. Colors are used as a visual aid to distinguish dependencies; all arrow segments with the same color are connected. Finally, components are depicted via dotted boxes with their sequence index in the top left corner. We have 13 components and see that components $P_{12}$ and $P_{13}$ involve recursive rules because they depend on themselves. This recursion is also reflected in the way operators proceed, in the presence of recursion it takes multiple applications of an operator before a fixed point is reached as can be seen in Example 3.2.5 below.

With an instantiation sequence at hand, we can ground subprograms successively taking into account certain and possible atoms derived previously. Accordingly, we extend the operators introduced above with context information. In what follows, we append letter ‘C’ to names of interpretations having a contextual nature.

Given a program $P$, we define the one-step provability operator of $P$ relative to contextual interpretation $IC$ for an interpretation $I$ as

$$T_P^{IC}(I) = T_P(IC \cup I),$$
the stable operator for $P$ relative to contextual interpretation $IC$ for an interpretation $J$ as

$$S^IC_P(J) = \text{lfp}_{\subseteq}(T^IC_P J),$$

the well-founded operator of $P$ relative to contextual interpretation $(IC, JC)$ for an interpretation $(I, J)$ as

$$W^ICJC_P(I, J) = (S^IC_P(J), S^JC_P(I)),$$

and the well-founded model of $P$ relative to contextual interpretation $(IC, JC)$ as

$$WM^ICJC_P(P) = \text{lfp}_{\leq P}(W^ICJC_P P).$$

To characterize our grounding algorithms, we are only interested in contextual well-founded models of subprograms. We do not consider other kinds of contextual models here.

**Example 3.2.5.** Let us use component $P_{12}$ from Figure 3.2.1, the set $F$ from Example 3.1.3, and the contextual interpretations $IC = JC = F$

$$\cup \{e(1, 1), e(2, 2)\} \quad \text{via} \ r_1$$

$$\cup \{ne(1, 2), ne(2, 1)\} \quad \text{via} \ r_2$$

$$\cup \{o(1, 1, 0), o(2, 1, 0)\} \quad \text{via} \ r_3.$$

With this, we compute the certain atoms $I = S^IC_P(A)$ via the smallest fixed point of $T^IC_P A$. Observe that the reduct with the atom base $A$ discards all instances of rules $r_4, r_5$, and $r_7$. We cannot use rule $r_6$ either because atoms over predicate $m$ are neither contained in the contextual interpretation nor derivable by any rules. We obtain that

$$I = T^IC_P A(\emptyset) = \{\text{no}(1, 2, 0), \text{no}(2, 2, 0)\} \quad \text{via} \ r_8$$

is the least fixed point of $T^IC_P A$.

Next, let us compute the possible atoms $J = S^JC_P(\emptyset)$ via the smallest fixed point of $T^JC_P J$. Unlike above, the reduct with the empty set removes the negative bodies without discarding any rules. Thus, we can now use rules $r_4, r_5$, and $r_7$ to derive atoms. We obtain the interpretations

$$J_0 = T^JC_P A(\emptyset) = I$$

$$\cup \{m(1, 2, 1), m(2, 2, 1)\} \quad \text{via} \ r_4$$

$$\cup \{nm(1, 2, 1), nm(2, 2, 1)\} \quad \text{via} \ r_5$$

$$\cup \{o(1, 1, 1), o(2, 1, 1)\} \quad \text{via} \ r_7.$$

$$J_1 = T^JC_P J_0(\emptyset) = J_0$$

$$\cup \{m(1, 2, 2), m(2, 2, 2)\} \quad \text{via} \ r_4$$

$$\cup \{nm(1, 2, 2), nm(2, 2, 2)\} \quad \text{via} \ r_5.$$
3.2. GROUNDING

\[ \cup \{ o(1, 2, 1), o(2, 2, 1) \} \quad \text{via } r_6 \]
\[ \cup \{ o(1, 1, 2), o(2, 1, 2) \} \quad \text{via } r_7 \]
\[ \cup \{ no(1, 2, 1), no(2, 2, 1) \} \quad \text{via } r_8, \]

\[ J_2 = T_{P_{12}^J}^J (J_1) = J_1 \]
\[ \cup \{ m(1, 1, 2), m(2, 1, 2) \} \quad \text{via } r_1 \]
\[ \cup \{ nm(1, 1, 2), nm(2, 1, 2) \} \quad \text{via } r_5 \]
\[ \cup \{ o(1, 2, 2), o(2, 2, 2) \} \quad \text{via } r_6 \]
\[ \cup \{ no(1, 1, 1), no(2, 1, 1), no(1, 2, 2), no(2, 2, 2) \} \quad \text{via } r_8, \]

\[ J_3 = T_{P_{12}^J}^J (J_2) = J_2 \]
\[ \cup \{ no(1, 1, 2), no(2, 1, 2) \} \quad \text{via } r_8. \]

The set \( J = J_3 \) is the least fixed point of \( T_{P_{12}^J}^J \).

Observe that we have just computed \((I, J) = W_{P_{12}^J, J}^J (\emptyset, A)\). This is also the least fixed point of the well-founded operator \( W_{P_{12}^J, J}^J \). We obtain that \((I, J) = WM_{P_{12}^J, J}^J (P_{12})\) is the well-founded model of \( P_{12} \) in the context \((IC, JC)\).

We now apply the contextual well-founded operator to an instantiation sequence to iteratively compute the well-founded model of the whole program. We define the well-founded model of instantiation sequence \((P_i)_{i \in \mathbb{I}}\) for program \( P \) as \( WM((P_i)_{i \in \mathbb{I}}) = \bigsqcup_{i \in \mathbb{I}} (I_i, J_i) \) where

\[ (IC_i, JC_i) = \bigsqcup_{i<j} (I_i, J_i) \quad \text{and} \]
\[ (I_i, J_i) = WM_{IC_i, JC_i}^J (P_i). \]

We shown in Section 2.2 that the well-founded model \( WM((P_i)_{i \in \mathbb{I}}) \) of the sequence is equivalent to the well-founded \( WM(P) \) of the whole program.

**Example 3.2.6.** Next, we show how the approximate model of the sequence \((P_i)_{1 \leq i \leq 13}\) is calculated. We give below the intermediate models \((I_i, J_i)\) as in Equation (2) for
Observe that the union of all models corresponds to the well-founded model in Example 3.2.2. Furthermore, note that we have seen in Example 3.2.5 how to compute the contextual well-founded model of component \( P_{12} \). The contextual interpretation \( \bigcup_{1 \leq i \leq 11} (I_i, J_i) \) in Equation (1) and the contextual well-founded model \((I_{12}, J_{12})\) in Equation (2) correspond to the contextual interpretation/well-founded model used/computed in the example, respectively.

In practice, grounding algorithms cannot compute the well-founded model on-the-fly but rather compute an approximation. We define the approximate model relative to contextual interpretation \((IC, JC)\) of a program \( P \) as \( AM^{IC,JC}(P) = (I, J) \) where

\[
P' = \{ r \in P \mid \text{pred}(B(r)) \cap \text{pred}(H(P)) = \emptyset \},
\]

\[
I = S_{P'}^{IC}(JC), \text{ and}
\]

\[
J = S_{P'}^{JC}(IC \cup I).
\]

**Example 3.2.7.** The contextual approximate model \( AM^{IC,JC}(P_{12}) \) is equivalent to the contextual well-founded model \( WM^{IC,JC}(P_{12}) \) in Example 3.2.5. This is because the reduct \( P_{12}^A \) is equivalent to the program \( P_{12}' \) obtained from \( P_{12} \) as in (3) and the well-founded model is computed with just one application of the well-founded operator applied to the least precise interpretation \((\emptyset, A)\).

Analogous to above, we can now apply this contextual operator to a sequence of programs. We define the approximate model of instantiation sequence \((P_i)_{i \in I}\) for program \( P \) as \( AM((P_i)_{i \in I}) = \bigcup_{i \in I} (I_i, J_i) \) where

\[
(IC_i, JC_i) = \bigcup_{i < j} (I_i, J_i) \text{ and}
\]

\[
(I_i, J_i) = AM^{IC_i,JC_i}(P_i).
\]

Since the approximate model is restricted to two applications of the stable operator, it is less precise than the well-founded model in general. We obtain that \( AM((P_i)_{i \in I}) \leq_p WM((P_i)_{i \in I}) \). For stratified programs, where no strongly connected component depends
negatively on itself, we obtain that \( AM((P_i)_{i \in I}) = WM((P_i)_{i \in I}) \). In fact, the approximate model is total in this case, that is, the certain and possible atoms of the model coincide. As a consequence, the grounding component of clingo can completely evaluate such programs. Finally, since the approximate model is the same for all instantiation sequences of a program, we define the approximate model of a program \( P \) as \( AM(P) = AM((P_i)_{i \in I}) \) using an arbitrary instantiation sequence of \( P \).

**Example 3.2.8.** We have already seen in Example 3.2.7 that the contextual approximate and well-founded models are equivalent for component \( P_{12} \). In fact, this is the case for all intermediate models when calculating the respective models of the sequence. Thus, the approximate and well-founded models of the sequence are equivalent, too.

Before we can turn to grounding algorithms, we restrict the class of programs to so-called safe programs. A program is safe if all its rules are safe; a rule is safe if all variables occurring in it also occur in its positive body. Our algorithms operate on safe rules and programs only. Furthermore, this allows us to restrict the signature of a program to the symbols appearing in a program; adding further symbols to this signature does not change the semantics of the program in the sense that the stable and well-founded models are the same as the ones of the program with the restricted signature [25].

**Example 3.2.9.** Our example program in Listings 3.1.1 and 3.1.2 is safe.

### 3.2.2 Algorithms

A substitution over signature \( \Sigma \) is a mapping from the variables in \( \Sigma \) to the constants and variables in \( \Sigma \). We use \( \iota \) to denote the substitution mapping each variable to itself. The result of applying a substitution \( \sigma \) to an expression \( e \), written \( e\sigma \), is the expression obtained by replacing all occurrences of each variable \( v \) in \( e \) by \( \sigma(v) \). The composition of two substitutions \( \sigma \) and \( \theta \) is the substitution \( \sigma \circ \theta \) where \((\sigma \circ \theta)(v) = \theta(\sigma(v))\).

In the following, we are interested in substitutions that step-by-step replace variables in body literals to obtain rule instances. A substitution \( \sigma \) is a matcher of expression \( e \) to variable-free expression \( g \), if \( e\sigma = g \) and \( \sigma \) maps all variables not occurring in \( e \) to itself.

Note that there is at most one matcher of \( e \) to \( g \). We let

\[
\text{match}(e,g) = \begin{cases} \{\sigma\} & \text{if there is a matcher } \sigma \text{ of } e \text{ to } g, \\ \emptyset & \text{otherwise.} \end{cases}
\]

When grounding rules, we look for matches of body literals in the atoms accumulated so far. The latter is captured by a four-valued interpretation to distinguish certain atoms among the possible ones. Given a substitution \( \sigma \), a literal \( l \), and an interpretation \((I,J)\), we define the set of matches for \( l \) in \((I,J)\) w.r.t. \( \sigma \) as

\[
\text{Matches}_{I,J}^l(\sigma) = \begin{cases} \{\sigma\} & \text{if } a = \neg l, \\ \{\sigma \circ \sigma' | a \in J, \sigma' \in \text{match}(l\sigma,a)\} & \text{otherwise}. \end{cases}
\]

In this way, positive body literals yield a (possibly empty) set of substitutions refining the one at hand, while negative literals are only considered when ground and then act as a test on the given substitution.

Function \text{GroundRule} for rule instantiation is given in Algorithm 3.1. It takes a substitution \( \sigma \) and a set \( L \) of literals and yields a set of instances of a safe normal rule
function $\text{GroundRule}_{r,I,J}^{l,f,J'}(\sigma, L)$:

1. if $L \neq \emptyset$:
   
   2. $(G, l) \leftarrow (\emptyset, \text{Select}_\sigma(L))$

   3. for each $\sigma'$ in $\text{Matches}_{I,J}^l(\sigma)$:
      
      4. $G \leftarrow G \cup \text{GroundRule}_{r,I,J'}^{l,f,J'}(\sigma', L \setminus \{l\})$

   5. return $G$

6. else if $f = t$ or $B(r\sigma)^+ \nsubseteq J'$:
      
      7. return $\{r\sigma\}$

8. else:
      
      9. return $\emptyset$

Algorithm 3.1. Grounding rules

$r$, passed as a parameter; it is called from Algorithm 3.2 with the identity substitution
and the body literals $B(r)$ of $r$. The other parameters consist of an interpretation
$(I, J)$ comprising the set of possibly derivable atoms along with the certain ones, an
interpretation $J'$ reflecting the previous value of $J$, and a Boolean flag $f$ used to avoid
duplicate rule instances in consecutive calls to Algorithm 3.1. The idea is to extend the
current substitution in Lines 4 to 5 until we obtain a substitution $\sigma$ that induces an
instance $r\sigma$ of rule $r$. To this end, $\text{Select}_\sigma(L)$ picks for each call some literal $l \in L$ such
that $l \in L^+$ or $l\sigma$ is ground. That is, it yields either a positive body literal or a ground
negative body literal, as needed for computing $\text{Matches}_{I,J}^l(\sigma)$. Whenever an application
of $\text{Matches}$ for the selected literal in $B(r)$ results in a non-empty set of substitutions,
the function is called recursively for each such substitution. The function then returns a
set of all rule instances obtained from the recursive calls in Line 8. (Note that we refrain
from applying any simplifications to rule instances and rather leave them intact to obtain
more direct formal characterizations of the results of our grounding algorithms.) The recursion terminates if the set of remaining body literals is empty, in which case, we
obtain a rule instance $r\sigma$. The test $B(r\sigma)^+ \nsubseteq J'$ in Line 7 makes sure that the instance
is discarded if it was already obtained by a previous invocation $\text{GroundRule}$ where the
flag $f$ ensures that all instances are gathered in the first iteration. This is relevant for
recursive rules and reflects the approach of semi-naive database evaluation $^2$. We obtain
that

- $\{r\}^{l,J} = \text{GroundRule}_{r,I,J}(l, B(r))$, and
- $\{r\}^{l,J} = \{r\}^{l,J'} \cup \text{GroundRule}_{r,f,J'}^{l,J}(l, B(r))$.

Thus, calls to $\text{GroundRule}$ amount to applications of the one-step provability operator
because $H(\{r\}^{l,J}) = T_{p_l}(J)$. This can easily be seen comparing the definition of of the
program simplification and the one-step-operator, which gather rules and rule heads
under the same condition.

Example 3.2.10. Next, we show how the two calls $\text{GroundRule}_{r,JC}^{IC,JC}(\sigma, B(r))$ and
$\text{GroundRule}_{r,IC,J'C}^{IC,JC,J'}(\sigma, B(r))$ proceed when called with interpretation $J_0$ and contextual
interpretation $(IC, JC)$ as given in Example 3.2.5 using the rules from Listing 3.1.2.
The execution of the two calls is illustrated in Table 1. The header above the double line contains the literals of the body of rule \( r_4 \) in the first four columns in the order they are selected for grounding in Line 3, as well as the head atom of the rule in the last column. Rows 2 and 3 correspond to the first and rows 4 to 7 to the second call to \texttt{GroundRule}. Both groups of rows are separated by a horizontal line. A group of matches is connected via vertical lines and is iterated in the loop in Lines 4 to 5. Recursive calls in Line 5 to \texttt{GroundRule} are indicated via horizontal arrows. We use the \( \times \) symbol to indicate that a rule instance has been discarded due to the test in Line 7. We can read off a rule instance by following arrows from the left to the rightmost column selecting one atom out of a group of matches.

Observe that the first call produces all instances of \( r_4 \) with atoms over predicate \( m \) as head that are contained in \( J_0 \) from Example 3.2.5. Similarly, the second call produces all instances with atoms over predicate \( m \) that are contained in \( J_1 \) from Example 3.2.5 not yet included in \( J_0 \). We can call function \texttt{GroundRule} for the remaining rules \( r_5 \) to \( r_8 \) to compute all sets \( J_i \) from Example 3.2.5.

Finally, note that we could discard the matches \( o(1,1,0) \) and \( o(2,1,0) \) in the second call right away because no atoms over the remaining predicates in the positive body of the rule have been added to \( J_0 \). The grounding algorithm of clingo is optimized to avoid processing such matches.

Let us now turn to grounding components of instantiation sequences in Algorithm 3.2. The function \texttt{GroundComponent} takes a program \( P \) along with two sets \( IC \) and \( JC \) of ground atoms as input. Intuitively, \( P \) is a component in an instantiation sequence and \( IC \) and \( JC \) form a four-valued interpretation \( (IC, JC) \) comprising the certain and possible atoms gathered while grounding previous components (although their roles get reversed in Algorithm 3.3). After variable initialization, \texttt{GroundComponent} loops over consecutive rule instantiations in \( G \) until no more possible atoms are obtained. In this case, it returns in Line 6 the set of obtained rule instances. In more detail, the program \( P \) is instantiated in Line 4, at which point \( J \) holds the possible atoms derived so far and \( J' \) its previous
function GroundComponent\((P, IC, JC)\):
\[(G, f, J', J) \leftarrow (\emptyset, \top, \emptyset, \emptyset)\]

while \(f = \top\) or \(J' \neq J\):
\[G \leftarrow G \cup \bigcup_{r \in P} \text{GroundRule}_{IC, JC, JJ}^r(f, J, B(r))\]
\[(f, J', J) \leftarrow (f, J, J \cup H(G))\]

return \(G\)

Algorithm 3.2. Grounding components

function Ground\((P)\):
let \((P_i)_{i \in I}\) be an instantiation sequence for \(P\)
\[(F, G) \leftarrow (\emptyset, \emptyset)\]
for each \(i\) in \(I\):
\[P'_i \leftarrow \{ r \in P_i \mid \text{pred}(H(P_i)) \cap \text{pred}(B(r)) = \emptyset \}\]
\[F \leftarrow F \cup \text{GroundComponent}(P'_i, H(G), H(F))\]
\[G \leftarrow G \cup \text{GroundComponent}(P_i, H(F), H(G))\]

return \(G\)

Algorithm 3.3. Grounding programs

value. For the next iteration, \(J\) is augmented in Line 5 with all rule heads in \(G\) and the flag \(f\) is set to false. Recall that the purpose of \(f\) is to ensure that initially all rules are grounded. In subsequent iterations, duplicates are omitted by setting the flag to false and filtering rules whose positive bodies are a subset of the atoms \(J'\) used in previous iterations. Given \(J = S_P^I JC\) and \(G = \text{GroundComponent}(P, IC, JC)\), we have \(H(G) = J\) and \(G = P^{IC, JC, JJ}\). Thus, function \(\text{GroundComponent}\) reflects the application of the stable model operator relative to contextual interpretation \((JC, IC)\).

Example 3.2.11. We have seen in Example 3.2.10 that four successive calls to \(\text{GroundRule}\) for each rule in component \(P_{12}\) produce the rule instances whose heads correspond to interpretation \(J\) from Example 3.2.5. Function \(\text{GroundComponent}\) called with context \((IC, JC)\) calls \(\text{GroundRule}\) successively to obtain exactly those rule instances.

Note that the loop performs one unnecessary iteration. The last interpretation \(J_3\) only adds atoms over predicate \(no\), which cannot be used to provide further rule instances because atoms over \(no\) only occur negatively in rule bodies. The clingo system avoids such iterations by avoiding calls to \(\text{GroundRule}\) if there is no positive literal in a rule that can provide at least one match including an atom from the previous iteration.

Finally, Algorithm 3.3 grounds programs by iterating over the components of one of its instantiation sequences. Just as Algorithm 3.2 reflects the application of a stable operator, function \(\text{Ground}\) follows the definition of an approximate model. At first, we construct program \(P'\) from the component at hand by stripping rules involved in a negative cycle; this is the same program as in (3). Then, rule instances providing certain atoms are computed in Line 6; the head atoms of those instances correspond to interpretation \(I\) in (4). Similarly, rule instances providing possible atoms are computed in Line 7; the
head atoms correspond to interpretation $J$ in (5). Accordingly, the whole iteration aligns with the approximate model. We obtain our main result that $\text{Ground}(P) = P^{AM}(P)$.

**Example 3.2.12.** In Example 3.2.11, we have seen that the atoms obtained by the contextual stable operator correspond to the rule heads produced by function function $\text{GroundComponent}$. Furthermore, in Example 3.2.7, we have seen the intermediate models computed when calculating the approximate models via calls to the contextual stable operator. The rule instances produced to obtain these intermediate models correspond to the output of our grounding algorithm. They comprise relevant instances in the sense that all atoms appearing positively in body literals are possible and atoms appearing in negative body literals are not certain.

Listing 3.2.1 shows the rule instances as output by clingo, when the plain text output via option $\text{--text}$ is requested. First, we see that clingo removes literals from rule bodies. Given an intermediate model $(I,J)$, it removes positive body literals contained in $I$ (for example in Line 5), and negative body literals not contained in $J$ (for example in Line 7). This simplification can be disabled using option $\text{--facts-only}$ to obtain an output closer to the one described in this section. Second, we see that clingo further refines components. The component $P_{12} = \{r_4, r_5, r_6, r_7, r_8\}$ from Example 3.2.4 is refined further into components $\{r_4, r_6, r_7\}$, $\{r_5\}$, and $\{r_8\}$. Observe that atoms over predicate $nn$ as derived by rule $r_5$ only occur negatively in rule bodies in $P_{12}$. This means that the rule cannot be used to derive certain atoms because it is removed from the component in Line 5. Since there are no certain atoms over predicate $nn$, negative literals over $nn$ always match and $\text{GroundRule}$ never discards rule instances because of them. Thus, we can as well ground the rule later.

Otherwise, observe that the rule instances directly correspond to the ones produced by Algorithm 3.3. We see, for example, that the rule heads in Lines 11 to 23 directly correspond to the ones reported in Example 3.2.5 (excluding the ones in components $\{r_5\}$ and $\{r_8\}$, which are grounded later). The rules in Lines 11 to 12 correspond to $J_0$, the rules in Lines 13 to 17 to $J_1$, the rules in Lines 18 to 21 to $J_2$, and the rules in Lines 22 to 23 to $J_3$. Even though no new atoms are added in the last iteration of the loop in $\text{GroundComponent}$, it still gathers rule instances and then exits the loop. In practice, the refined instantiation sequence used in clingo can reduce the peak memory used by the system because smaller components are considered at a time, and also in some cases provide more precise approximate models.

**3.2.3 Summary.** In Section 2.2, we provide the first comprehensive elaboration of the theoretical foundations of grounding. We follow an operator based characterization. These operators directly correspond to the recursion/loops in our grounding algorithms and facilitate proving their correctness. In this section, we have shown this connection for a simple class of programs. To cover the larger class of aggregate programs with aggregates and uninterpreted functions\(^2\), we rely on programs with infinitary formulas as rule bodies. This is important because the atom base associated with such programs is infinite in general, which results in infinitary formulas for aggregates. Yet, such formulas

\(^2\)In practice, we rarely nest uninterpreted functions in programs but many programs use arithmetic operations. It is relatively straightforward to extend our results for programs with uninterpreted functions to programs with arithmetic operations.
% facts
trans(0,1). trans(1,2). peg(1). peg(2).
init(1,1). init(2,1). disc(1). disc(2).

% component \{r_1\}
e(1,1).
% component \{r_2\}
e(2,2).
% component \{r_3\}
e(1,2).

% component \{r_4, r_5, r_7\}
o(1,1,0). o(2,1,0).
o(2,1,1). o(1,1,1). o(2,1,1).
o(2,2,1). o(1,2,1). o(2,2,1).
o(1,2,1). m(1,2,1). o(2,2,1).
o(1,2,2). m(1,2,2). o(1,2,2).
o(2,2,2). m(2,2,2). o(2,2,2).

% component \{r_8\}
o(2,1,1). o(1,1,1). o(2,1,1).
o(2,2,1). o(1,2,1). o(2,2,1).
o(2,2,2). o(1,1,2). o(2,2,2).
o(1,2,1). m(1,2,1). o(2,2,1).
o(1,2,2). m(1,2,2). o(1,2,2).

% component \{r_9\}
o(1,1,1). o(2,1,1). o(2,1,1).
o(1,1,2). o(2,2,2). o(2,1,2).
o(2,2,2). o(1,2,2). o(2,2,2).

% component \{r_6\}
o(1,1,2). o(1,2,1). o(1,2,1).
o(2,1,2). o(2,2,2). o(2,2,2).

% component \{r_5\}
of(2,1,1), o(2,1,1), not f.
of(2,1,1), o(1,1,1), not f.
of(1,1,2), o(1,1,2), not f.
of(2,2,1), o(2,1,1), not f.
of(1,2,1), o(1,2,1), not f.
of(2,2,1), o(2,2,2), not f.
of(1,2,2), o(1,2,2), not f.

Listing 3.2.1. Clingo output for simplified ToH example
3.3. MULTI-SHOT SOLVING

3.3.1. MULTI-SHOT SOLVING

It is no longer possible to obtain operators in the same way as we presented above for our simplified language. Instead, we borrow the operators in [139] defined for an alternative semantics and define the well-founded and approximate model in terms of them. Interestingly, the fixed points of the stable operator no longer correspond to the stable models of our target semantics. Thus, we identify a restricted class of programs powerful enough to capture aggregate programs. The well-founded and approximate models approximate the stable models of this class. For our simplified programs, we have seen that we can simplify a program using the approximate model to obtain a set of relevant rule instances. The same is true for aggregate programs but we have to introduce an additional complementary simplification that restricts the size of the infinitary formulas for aggregates. It turns out, that the grounding algorithms produce a finite set of equivalent rules for an aggregate program whenever the approximate model of the program is finite. Finally, let us note that grounding algorithms for aggregate programs build on top of the algorithms presented in this section. In fact, Algorithms 3.1 and 3.3 are left untouched from a high level point of view. Only Algorithm 3.2 is extended, adding additional steps to ground aggregates. The first step decomposes aggregates into rules without aggregates similar to the ones presented in this section. These rules are then grounded using Algorithm 3.1 and the aggregates are reassembled from the obtained rule instances in the last step.

The approach to grounding presented in this section follows the traditional two-phase approach of grounding and solving as implemented by mainstream ASP systems [55, 132]. It completely separates the grounding process from the subsequent solving. This can lead to grounding becoming a bottleneck when the grounding algorithms produce too many instances [41]. An alternative way to tackle this problem is lazy grounding and solving [112, 124, 144], which grounds rules on-the-fly while solving. Potentially, much fewer rule instances are produced during this process. However, it becomes much harder to adapt existing solving technology to this setting. Another interesting approach is to move part of the grounding process to solving. It has been shown in [17] that normal logic programs with fixed predicate arities can be translated into disjunctive logic programs that have polynomial size groundings. This approach can drastically reduce the grounding effort at the expense of having to solve a second level problem afterward.

3.3 Multi-shot solving

Standard ASP solving follows a one-shot approach consisting of first grounding and then solving a program. This is also reflected by the available systems. For example, take lparse [135] and smodels [121], the former grounds a program and then passes the grounded program to the latter to compute stable models. Even monolithic systems like dlv [113] and clingo [74] (up to version 3) proceed in the same way, passing the grounded program internally from the grounding to the solving component. The iclingo [75] and oclingo [68] systems where the first to deviate from this rigid reasoning process. The former was designed to solve planning problems by gradually extending a program w.r.t. an increasing bound on the solution size. The latter took this approach one step further by providing facilities to react to real-time events during the reasoning process. Yet, both systems follow a fixed control flow that evades fine-grained user control. Beyond this, however, there is substantial need for specifying flexible reasoning processes. For
example, when it comes to interactions with an environment (as in robotics or with users), advanced search (as in multi-objective optimization, planning, theory solving, or heuristic search), or recurrent query answering (as in hardware analysis and testing or stream processing). Common to all these advanced forms of reasoning is that the problem specification evolves during the respective reasoning processes, either because data or constraints are added, deleted, or replaced.

Section 2.3 comprises the work in [80] presenting the multi-shot solving capabilities of the clingo system. Multi-shot solving allows us to address complex reasoning tasks involving continuously changing programs. Such changes include adding rules to or deleting rules from a logic program as well as setting truth values of input atoms to react to external input. Here, we first introduce the necessary background building upon the theoretical foundations of grounding presented in Section 3.2. We illustrate this material using the simplified ToH problem from Section 3.1. In the second part, we present a simplified state-based formalization of clingo’s Application Programmable Interface (API) and show how to use it to solve the full ToH problem as given in Chapter 1. The main advantage is that we no longer have to bound the plan length in advance but incrementally extend a program until a solution is found.

### 3.3.1 Background.

We use the notation for programs and related concepts from Section 3.2. We first define concepts for ground programs and, afterward, show how to use our grounding algorithms to apply them to non-ground programs.

Given a ground program $P$, we say that an atom $h \in A$ depends positively on an atom $b \in A$ if there is a rule $r \in \Gamma(P)$ such that $h = H(r)$ and $b \in B(r)^+$. The strongly connected atom components of a program $P$ are the equivalence classes of the positive atom dependency relation.

A module $P$ is a triple $(P, JI, JO)$ where $P$ is a ground program, and $JI$ and $JO$ are disjoint sets of ground input and output atoms as indicated by the suffixes $I$ and $O$, respectively. Furthermore, we require that the body atoms $B(P)^\pm \subseteq JI \cup JO$ (6) are either output or input atoms and that all head atoms $H(P) \subseteq JO$ (7) are output atoms. Observe that because $JI$ and $JO$ are disjoint, we also obtain that the head atoms cannot be input atoms, that is, $H(P) \cap JI = \emptyset$.

An interpretation $X$ is a stable model of a module $(P, JI, JO)$ if $X$ is a stable model of $P \cup \{a \leftarrow | a \in JI \cap X\}$. Two modules $(P_1, JI_1, JO_1)$ and $(P_2, JI_2, JO_2)$ are compositional if

(8) $JO_1 \cap JO_2 = \emptyset$

(9) $JO_1 \cap C = \emptyset$ or $JO_2 \cap C = \emptyset$

for every strongly connected atom component $C$ of $P_1 \cup P_2$. In other words, all rules defining an atom must belong to the same module and positive recursion must be within modules—not among them. Given two compositional modules $P_1 = (P_1, JI_1, JO_1)$
and $P_2 = (P_2, JI_2, JO_2)$, we define their join $P_1 \Join P_2$ as

$$
(P_1 \cup P_2, (JI_1 \setminus JO_2) \cup (JI_2 \setminus JO_1), JO_1 \cup JO_2).
$$

The module theorem [123] shows that, for compositional modules, an interpretation $X$ is a stable model of $P_1 \Join P_2$ iff $X = X_1 \cup X_2$ for stable models $X_1$ and $X_2$ of $P_1$ and $P_2$, respectively, such that $X_1 \cap (JI_1 \cup JO_2) = X_2 \cap (JI_2 \cup JO_1)$. Finally, we assume that the join operator is left associative to avoid writing parenthesis, that is, $P_1 \Join P_2 \Join P_3 = (P_1 \Join P_2) \Join P_3$ for modules $P_1$, $P_2$, and $P_3$.

**Example 3.3.1.** Taking up the simplified ToH example from Section 3.2, we use the set

$$
F = \{ o(1, 1, 0), ne(1, 2), step(0, 1),
\quad o(1, 2, 1), ne(2, 1), step(1, 2) \}
$$

of atoms to define the module

$$
P_0 = (\{ a \leftarrow \mid a \in F \}, \emptyset, F).
$$

Note that $F$ is a subset of the certain atoms as given in Example 3.2.2. For the purpose of this example, we consider the instances

$$(g_{4,1}) \quad m(1, 2, 1) \leftarrow \neg nm(1, 2, 1), o(1, 1, 0), ne(1, 2), step(0, 1)
\quad (g_{5,1}) \quad nm(1, 2, 1) \leftarrow \neg m(1, 2, 1), o(1, 1, 0), ne(1, 2), step(0, 1)
\quad (g_{6,1}) \quad o(1, 2, 1) \leftarrow m(1, 2, 1)
\quad (g_{4,2}) \quad m(1, 1, 2) \leftarrow \neg nm(1, 1, 2), o(1, 2, 0), ne(2, 1), step(1, 2)
\quad (g_{5,2}) \quad nm(1, 1, 2) \leftarrow \neg m(1, 1, 2), o(1, 2, 0), ne(2, 1), step(1, 2)
\quad (g_{6,2}) \quad o(1, 1, 2) \leftarrow m(1, 1, 2)
$$

of rules $r_4$, $r_5$, and $r_6$ from Example 3.1.1, and the modules

$$
P_1 = \{ \{ g_{4,1}, g_{5,1}, g_{6,1} \},
\quad \{ o(1, 1, 0), ne(1, 2), step(0, 1) \},
\quad \{ m(1, 2, 1), nm(1, 2, 1), o(1, 2, 1) \} \} \text{ and }
\quad P_2 = \{ \{ g_{4,2}, g_{5,2}, g_{6,2} \},
\quad \{ o(1, 2, 1), ne(2, 1), step(1, 2) \},
\quad \{ m(1, 1, 2), nm(1, 1, 2), o(1, 1, 2) \} \}.
$$

Observe that we can compose modules $P_0 \Join P_1 \Join P_2$, that is, property (8) holds because the outputs of all three modules are disjoint, and property (9) holds because there is no positive recursion in our example program.
Thus, we can use the module theorem to compute the stable models of the join, which are simply the stable models of the union $P_0 \cup P_1 \cup P_1$ because $P_0$ has no inputs:

$$X_1 = F \cup \{nm(1,2,1)\},$$
$$X_2 = F \cup \{m(1,2,1), o(1,2,1), nm(1,1,2)\}, \text{ and}$$
$$X_3 = F \cup \{m(1,2,1), o(1,2,1), m(1,1,2), o(1,1,2)\}.$$

We further develop this example in the following and show that solutions to the ToH problem up to a time step $i$ can be captured by modules $P_i$ starting from a module that represents the initial state $P_0$. From a high level point of view, incremental solving proceeds as follows: we ground a module for a program (capturing a time step in our example), extend the previous one with it, and then compute the stable models of the extended module. Starting with the empty module $(\emptyset, \emptyset, \emptyset)$, this process is repeated until the desired set of stable models is obtained.

Next, we show how to ground and extend modules. We define the ground module for program $P$ relative to $G_1$ as

$$\Gamma M(G_1, P) = G_2$$

where

$$G_1 = (G_1, JI_1, JO_1),$$
$$G_2 = (\Gamma(P), JI_2, JO_2),$$
$$JI_2 = JO_1 \setminus H(\Gamma(P)),$$
$$JO_2 = A(\Gamma(P)) \setminus JI_2.$$ 

Observe that the modules $G_1$ and $G_2$ are not necessarily compositional. In our current setup, where the inputs of $G_2$ are solely given by the outputs of $G_1$, they are compositional whenever $JO_1 \cap H(\Gamma(P)) = \emptyset$.\footnote{The clingo system does not check if programs are compositional. It is left to the user to write programs that satisfy both properties. However, we see in Section 3.4, how to build applications on top of clingo’s multi-shot solving that are more user friendly in this regard.} If the modules are compositional, we obtain the module $G_1 \bowtie G_2$, which can be extended by grounding further programs. Note that for $G_1 = (\emptyset, \emptyset, \emptyset)$ the join $G_1 \bowtie G_2 = G_2$ is always compositional.

Next, we show how to use the well-founded model and program simplification from Section 3.2 to simplify modules. Let $P$ be a program, and

$$G_1 = (G_1, JI_1, JO_1)$$
$$G_2 = (G_2, JI_2, JO_2) = \Gamma M(G_1, P)$$

be modules such that $G_1$ and $G_2$ are compositional. Remember that we just defined the ground program $G_2$ to be the set $\Gamma(P)$ of all rule instances of $P$. We now show how to simplify program $G_2$ by means of the techniques developed in Section 3.2. Let

$$(I_1, J_1) = WM^{0,JI_1}(G_1)\text{ and}$$
$$(I_2, J_2) = WM^{JI_1,\cup(0,JI_1)}(G_2)$$
function Ground\( (P, IC, JC) \):

let \( (P_i)_{i \in I} \) be an instantiation sequence for \( P \)

\( (F, G) \leftarrow (\emptyset, \emptyset) \)

for each \( i \) in \( I \):

\( P'_i \leftarrow \{ r \in P_i \mid \text{pred}(H(P_i)) \cap \text{pred}(B(r^-)) = \emptyset \} \)

\( F \leftarrow F \cup \text{GroundComponent}(P'_i, H(G), H(F)) \)

\( G \leftarrow G \cup \text{GroundComponent}(P_i, H(F), H(G)) \)

return \( (G, H(F), H(G)) \)

Algorithm 3.4. Contextual grounding of programs

be contextual well-founded models of \( G_1 \) and \( G_2 \). Using these models, we simplify program \( G_2 \) by letting

\[ G'_2 = G_2^{(I_1,J_1) \sqcup (I_2,J_2) \sqcup (\emptyset,J_1)} \]

which we associate with the module

\[ G'_2 = (G'_2, J_1, \text{A}(G'_2) \setminus J_2) \].

Using the results from Section 3.2, we obtain that the modules \( G_2 \) and \( G'_2 \) as well as the joins \( G_1 \Join G_2 \) and \( G_1 \Join G'_2 \) have the same stable models.

We use the above construction as a template to modify function Ground in Algorithm 3.3 from Section 3.2 to extend modules. Instead of the well-founded model, the algorithm computes the less-precise approximate model. This is no problem because the above construction works with any approximation of the well-founded model. We now turn to the modified grounding algorithm in Algorithm 3.4. First, to provide the contextual interpretation, we add parameters \( IC \) and \( JC \) and ground components relative to them. Second, remember that the final ground programs assigned to variables \( F \) and \( G \) hold the approximate model \( (H(F), H(G)) \) of the input program. We return this model in addition to the grounded program in Line 8.

We now show how to ground a module using Algorithm 3.4. We define the contextual ground module for program \( P \) relative to module \( G_1 \) and contextual interpretation \( (IC, JC) \) as

\[ \text{GroundModule}^{IC,JC}(G_1, P) = (G_2, (I_2, J_2)) \]

where

\[ G_1 = (G_1, J_1, JO_1) \]

\( (G_2, I_2, J_2) = \text{Ground}(P, IC, JC \cup JI_1) \)

\[ G_2 = (G_2, JI_2, JO_2) \]

\( JI_2 = JO_1 \setminus H(G_2) \)

\( JO_2 = A(G_2) \setminus JI_2 \).

Next, we show when we can use the contextual ground module instead of the ground module of a program to extend a module in the sense that both joins have the same stable models. Let \( G_1 = (G_1, JI_1, JO_1) \) be a module, \( (IC, JC) \) be an interpretation,
(r'₄) \quad m(D, P, t) \leftarrow \neg nm(D, P, t), o(D, P', t'), ne(P', P)
(r'₅) \quad nm(D, P, t) \leftarrow \neg m(D, P, t), o(D, P', t'), ne(P', P)
(r'₆) \quad o(D, P, t) \leftarrow m(D, P, t)
(r'₇) \quad o(D, P, t) \leftarrow o(D, P', t'), \neg no(D, P, t)
(r'₈) \quad no(D, P, t) \leftarrow o(D, P', t), ne(P', P)
(r'₉) \quad f(t) \leftarrow \neg f(t), o(D, P, t), no(D, P, t)

Listing 3.3.1. Incremental simplified ToH encoding

\( P \) be a program, \( G_2 = \Gamma M(G_1, P) \), and \((G'_2, (I'_2, J'_2)) = \text{GroundModule}^{IC, JC}(G_1, P)\). If \((IC, JC) \leq_p WM^{IC, JC}(G_1)\) and \(G_1\) and \(G_2\) are compositional, then we get that

1. \( G_1 \ltimes G_2 \) and \( G_1 \ltimes G'_2 \) have the same stable models, and
2. \((IC, JC) \sqcup (I'_2, J'_2) \leq_p WM^{IC, JC}(G')\) where \((G', J', J'O') = G_1 \ltimes G'_2\).

Property (1) shows that we obtain equivalent modules using our grounding algorithm. Note that the modules can differ in their outputs, though. The outputs of \( G_1 \ltimes G_2 \) are a superset of the outputs of \( G_1 \ltimes G'_2 \) in general. Property (2) shows that we can use \((IC, JC) \sqcup (I'_2, J'_2)\) as contextual interpretation to further extend the module \( G_1 \ltimes G'_2 \).

To be able to ground compositional modules from a fixed set of programs, we define \textit{parametrizable} programs. To this end, we treat some of the constants occurring in programs as parameters, that is, we use a signature \( \Sigma = (\mathcal{C} \cup \mathcal{K}, \mathcal{P}, \mathcal{V}) \). Symbols in \( \mathcal{C}, \mathcal{P} \), and \( \mathcal{V} \) are used as before but the \textit{parameter} symbols in \( \mathcal{K} \) are replaced by constants in \( \mathcal{C} \) in a program before grounding it. Given a program \( P \) with parameters \( \mathcal{K} = \{k_1, \ldots, k_n\} \), we write \( P[k_1/c_1, \ldots, k_n/c_n] \) to obtain a program over signature \( (\mathcal{C}, \mathcal{P}, \mathcal{V}) \) from \( P \) by (simultaneously) replacing all occurrences of parameters \( k_i \) in it by constants \( c_i \in \mathcal{C} \). To keep the notation compact, we write \( P[k/c] \) instead of \( P[k_1/c_1, \ldots, k_n/c_n] \) for tuples \( k = (k_1, \ldots, k_n) \) and \( c = (c_1, \ldots, c_n) \) of parameters and symbols.

Example 3.3.2. We show how to use parametrizable programs to obtain compositional modules for each time step of our ToH example.

Let \( F \) be the rules in Listing 3.1.1 excluding the facts for transitions using predicate step. We omit them here because time points are captured by the parameters \( t \) for the current and \( t' \) for the previous time point. We let

\[ P_{\text{base}} = F \cup \{r_1, r_2, r_3\} \]

be a program without parameters using rules from Listing 3.1.2. The next two programs use rules from Listing 3.3.1, instead. We let

\[ P_{\text{step}} = \{r'_4, r'_5, r'_6\} \]

be a program with parameters \( t \) and \( t' \), and

\[ P_{\text{check}} = \{r'_8, r'_9\} \]

be a program with parameter \( t \).
3.3. MULTI-SHOT SOLVING

We can now incrementally ground modules for time steps. For time step 0, we let

\[(G_0, (I_0, J_0)) = \text{GroundModule}^{\emptyset, \emptyset}((\emptyset, \emptyset, \emptyset), P_{\text{base}} \cup P_{\text{check}}[t/0]) \text{ and}\]

\[P_0 = (\emptyset, \emptyset, \emptyset) \triangleright G_0 = G_0\]

by extending the empty module. We obtain that module

\[G_0 = (G_0, \emptyset, JQ_0)\]

has an empty set of inputs. Next, we inductively define modules for time steps 0 < \(i \leq 2\), letting

\[(IC_i, JC_i) = \bigsqcup_{j<i} (I_j, J_j),\]

\[(G_i, (I_i, J_i)) = \text{GroundModule}^{IC_i, JC_i}(P_{i-1}, P_{\text{step}}[t'(i-1), t/i] \cup P_{\text{check}}[t/i]), \text{ and}\]

\[P_i = P_{i-1} \triangleright G_i.\]

First, we verify that \(P_{i-1}\) and \(G_i = (G_i, JH_i, JO_i)\) are compositional. This is ensured in the way parameters \(t\) and \(t'\) are used in the programs \(P_{\text{step}}\) and \(P_{\text{check}}\). Remember that parameters \(t\) and \(t'\) refer to the current and previous time steps, respectively. Parameter \(t\) is used in all rule heads of the programs. Literals in rule bodies either use parameter \(t\) or \(t'\), or use predicates defined in the base program. Thus, rule heads define atoms for the current time step and rule bodies refer to the current or previous time steps. This ensures that the outputs of module \(P_{i-1}\) and the outputs of module \(G_i\) are disjoint. Hence, property \((8)\) holds for them. Property \((9)\) holds in our example, too, because our modules only use inputs from preceding modules, which prevents recursion among modules. In fact, we have \(JH_i = JO_0 \cup \cdots \cup JO_{i-1}\). We obtain that

\[P_i = (P_0 \cup \cdots \cup P_i, (\emptyset, \emptyset, JO_0 \cup \cdots \cup JO_i)).\]

Finally note that, excluding atoms over step, the stable models of program \(G_0 \cup \cdots \cup G_i\) correspond to the stable models \(X_0 \cup \cdots \cup X_i\) from Example 3.3.3, for horizons \(0 \leq i \leq 2\).

So far, we have seen how to obtain modules that use output atoms from previously grounded modules as input. Next, we show how to specify additional inputs to a module. We extend the signature \(\Sigma = (C, P \cup \{\epsilon\}, V)\) with a fresh nullary predicate \(\epsilon\). An extensible program over signature \(\Sigma\) can use atom \(\epsilon\) in positive rule bodies to mark rules whose only purpose is to provide input atoms; we refer to the heads of instances of such rules as external atoms.

We extend the definition of ground modules to extensible programs defining the ground module for extensible program \(P\) relative to module \(G_1\) and contextual interpretation \((IC, JC)\) as

\[\Gamma M(G_1, P) = G_2\]
where

\[
G_1 = (G_1, JI_1, JO_1), \\
G_2 = (G_2, JI_2, JO_2), \\
G_2 = \{g \in \Gamma(P) \mid \epsilon \notin B(g)\}, \\
JI_2 = (JO_1 \cup H(\Gamma(P))) \setminus H(G_2), \text{ and} \\
JO_2 = A(G_2) \setminus JI_2.
\]

\[
G_2 = (G_2, JI_2, JO_2), \\
G_2 = \{g \in G_2 \mid \epsilon \notin B(g)\}, \\
JI_2 = (JO_1 \cup H(G_2)) \setminus H(G_2), \text{ and} \\
JO_2 = A(G_2) \setminus JI_2.
\]

Note that all rules referring to \(\epsilon\) are stripped from \(\Gamma(P)\); such rules are only used to define the external atoms in \(H(\Gamma(P)) \setminus H(G_2)\).

As before, we can also use Algorithm 3.4 to extend a module. We extend the definition of the contextual ground modules to extensible programs, defining the contextual ground module for extensible program \(P\) relative to module \(G_1\) and contextual interpretation \((IC, JC)\) as

\[
\text{GroundModule}^{IC,JC}(G_1, P) = (G_2, (J_2, J_2))
\]

where

\[
G_1 = (G_1, JI_1, JO_1), \\
G_2 = (G_2, JI_2, JO_2), \\
(G_2', (J_2, J_2)) = \text{Ground}(P, IC, JC \cup JI_1 \cup \{\epsilon\}) \\
G_2 = \{g \in G_2' \mid \epsilon \notin B(g)\}, \\
JI_2 = (JO_1 \cup H(G_2')) \setminus H(G_2), \text{ and} \\
JO_2 = A(G_2) \setminus JI_2.
\]

Next, we shed some light on joins with ground modules and contextual ground modules for extensible programs. Let \(G_1 = (G_1, JI_1, JO_1)\) be a module, \((IC, JC)\) be an interpretation, \(P\) be an extensible program, \(G_2 = \Gamma M(G_1, P)\), and \((G_2', (J_2', J_2')) = \text{GroundModule}^{IC,JC}(G_1, P)\). If \((IC, JC) \leq_p \text{WM}^{B,JI}(G_1)\), and \(G_1\) and \(G_2\) are compositional, then we get

1. for interpretations \(X\) with \(X \cap (JI_1 \setminus JI_2') = \emptyset\), that \(X\) is a stable model of \(G_1 \times G_2\) if \(X\) is a stable model of \(G_1 \times G_2'\), where \(JI\) and \(JI'\) are the inputs of \(G_1 \times G_2\) and \(G_1 \times G_2'\), respectively, and

2. \((IC, JC) \cup (I_2', J_2') \leq_p \text{WM}^{B,JI'}(G')\) where \((G', JI', JO') = G_1 \times G_2'\).

For extensible programs, we no longer get that \(G_1 \times G_2'\) and \(G_1 \times G_2\) have the same stable models. This is because the inputs of \(G_2'\) are subject to grounding. Thus, the join \(G_1 \times G_2'\) might have fewer input atoms than \(G_1 \times G_2\). Property (1) still identifies a common set of stable models, namely, those in which the missing input atoms are false. Property (2) shows that we can use the interpretation \((IC, JC) \cup (I_2', J_2')\) as contextual interpretation to further extend the module \(G_1 \times G_2'\).
Example 3.3.3. We extend program $P_{\text{check}}$ from Example 3.3.2 with the following rules:

$$(r'_{10}) \quad \text{query}(t) \leftarrow \epsilon$$

$$(r'_{11}) \quad f(t) \leftarrow \neg f(t), \text{query}(t), \neg o(D, 2, t), \text{disc}(D)$$

Let $P'_i$ be the module constructed as in Example 3.3.2 using program $P_{\text{check}}$ extended with the above rules. The inputs of the module now additionally comprise the atoms $\text{query}(j)$ for $j \leq i$. In our setting, we are only interested in the stable models of $P'_i$ where the external atom $\text{query}(i)$ is true and the external atoms $\text{query}(j)$ are false for all $j < i$.

Module $P'_0$ has no stable models where atom $\text{query}(0)$ is true. Module $P'_1$ has exactly one stable model where atom $\text{query}(1)$ is true and atom $\text{query}(0)$ is false, namely, the model where all discs are moved from the first to the second peg in the first transition.

Remember that we defined stable models of modules in such a way that the input atoms can become true without any rules deriving them. The clingo system by default only computes stable models in which external atoms are false. However, its application programmable interface (API) can be used to let them flip freely or to pin them to concrete truth values. Using the module $P'_i = (G, JI, JO)$ from Example 3.3.3, we can use clingo’s API to fix the truth values of atoms in $JI$ as required. Thus, we can directly compute stable models for the program $G \cup \{ \text{query}(i) \leftarrow \}$. 

3.3.2 Multi-shot solving. We now turn to the (subset of the) clingo API for multi-shot solving. In the following, we use the Python language bindings; support for additional languages including C, C++, Lua, and more via third party projects are available. For capturing multi-shot solving, we account for system states holding information about the program kept within the system. To this end, we define a simple operational semantics based on system states and associated operations. Note that we present a simplified version of system states and their operations here. We refer to Section 2.3 for a full account.

A system state is a quadruple $(P, P, (IC, JC), A)$ where

- $P = (P_n)_{n \in \mathbb{N}}$ is a collection of parametrizable extensible programs,
- $P$ is a module,
- $(IC, JC)$ is an interpretation, and
- $A$ a set of atoms.

When solving with $P$, the input atoms are fixed according to the atoms in $A$, that is, input atoms contained in $A$ are set to true, and to false otherwise. System states can be created using function create, which reads a program from a list of files obtaining a collection of parametrizable extensible programs.

create(files) : $\mapsto (P, P, (IC, JC), A)$

for a collection $P$ of parametrizable extensible programs read from the given list of files where

- $P = (\emptyset, \emptyset, \emptyset)$,
- $(IC, JC) = (\emptyset, \emptyset)$, and
- $A = \emptyset$.

We do not detail how collections of programs are read from files and rather give an informal description in the following example.
### Listing 3.3.2. Incremental ToH encoding

```prolog
#program base.
% establish initial situation
on(D,P,0) :- init(D,P).

#program check(t).
% uniqueness of location: a disc can only be on one peg
-on(D,Q,t) :- on(D,P,t), peg(Q), P!=Q.

% ensure that the goal was reached
:- query(t), goal(D,P), not on(D,P,t).

#program step(t).
% choose discs to move
{ move(D,P,Q,t) } :- on(D,P,t-1), peg(Q), P!=Q.

% there must be at most one move per time step.
:- #count { D,P,Q: move(D,P,Q,t) } > 1.
% only the topmost disc can be moved
:- move(D,P,_,t), on(E,P,t-1), D>E.
% a disc can only be put on larger discs
:- move(D,_,Q,t), on(E,Q,t-1), D>E.

% effects: change the location of the moved disc
on(D,Q,t) :- move(D,_,Q,t).
% inertia: discs stay in place by default
on(D,P,t) :- on(D,P,t-1), not -on(D,P,t).

#show move/4.
```

**Example 3.3.4.** We now switch back to the ToH problem as presented in the introduction in Chapter 1 and give an encoding in Listing 3.3.2 to solve the problem using clingo’s multi-shot solving.

Note the additional directives starting with keyword `#program` in Lines 1, 5, and 12. Such program directives gather the following rules up to the next program directive or the end of the file. They give rise to parametrizable programs associated with the name and parameters following the keyword; parameters are enclosed in parenthesis, which can be omitted for empty parameter tuples. In our example, we have program $P_{\text{base}}$ without parameters, and programs $P_{\text{check}}$ and $P_{\text{step}}$ with parameter $t$. Note that unlike in Example 3.3.2, we do not need a parameter to refer to the previous time step because the clingo language supports arithmetic operations; we can simply use $t-1$, instead.

The incremental grounding of the ToH encoding is handled using the Python script in Listing 3.3.3, which is enclosed within `#script (python)` and `#end`. We delay a
3.3. MULTI-SHOT SOLVING

```python
#script (python)

from clingo.symbol import Number, Function

def main(ctl):
    step, ret, query = 0, None, None
    while ret is None or not ret.satisfiable:
        parts = [("check", [Number(step)])]
        if query is not None:
            ctl.release_external(query)
            parts.append("step", [Number(step)])
        else:
            parts.append("base", [])
        ctl.ground(parts)
        query = Function("query", [Number(step)])
        ctl.assign_external(query, True)
        ret, step = ctl.solve(), step + 1

#end.

#program check(t).
#external query(t).
```

Listing 3.3.3. Embedded Python code for incremental solving

description of this script until we have introduced the necessary operations to capture its semantics. Note, though, the `#external` directive in Line 21, which provides a convenient syntax to define external input atoms. It corresponds to the rule \( r_{10}^{P} \) from Example 3.3.3 and is part of the \( P_{\text{check}} \) program. We separate this directive from the encoding to be able to use the script to incrementally solve related problems.

Forgoing extended language constructs not supported by our simple language, we call create with files containing the instance, encoding, and script for the ToH problem in Listings 1.0.1a, 3.3.2, and 3.3.3 to obtain the collection of parametrizable extensible programs \( (P_n)_{n \in \mathbb{N}} \) for \( \mathbb{N} = \{\text{base}, \text{check}, \text{step}\} \). Note that the rules in Listing 1.0.1a are not subject to a program directive. Such rules are by default added to program \( P_{\text{base}} \). Thus, the initial program state is

\[
((P_n)_{n \in \mathbb{N}}, (\emptyset, \emptyset, \emptyset), (\emptyset, \emptyset), \emptyset).
\]

Note that the module associated with the state is still empty. We have to ground programs first to extend the module.

Function `ground` instantiates selected programs from the collection \( P \) for designated parameters. The resulting module is then used to extend the module of the current system state. We below assume that the join is always compositional: the `clingo` system
reports an error if condition (8) is violated and might report unexpected additional stable models if condition (9) is violated.\footnote{The solver translates a logic program into an internal representation for solving and afterwards discards the logic program to reduce the system’s memory consumption. The internal representation does not store enough information to detect cycles.}

\[ \text{ground}((n, c_n)_{n \in N'}) : (P_1, P_1, (IC_1, JC_1), A_1) \mapsto (P_2, P_2, (IC_2, JC_2), A_2) \]

for a collection of program names and tuples of constants to ground where

- \( P_2 \) is the collection of programs obtained from \( P_1 \) by extending the signatures of its programs with the constants in \( c_n \) for all \( n \in N' \),
- \((G, I, J) = \text{GroundModule}^{IC_1, JC_1}(P_1, \bigcup_{n \in N'} P_n[k_n/c_n]) \) where \( k_n \) are the parameters of program \( P_n \) and \((P_n)_{n \in N} = P_2 \),
- \( P_2 = P_1 \Join G \),
- \((IC_2, JC_2) = (IC_1, JC_1) \cup (I, J) \), and
- \( A_2 = A_1 \cap \text{JI} \) where \( \text{JI} \) are the inputs of \( P_2 \).

Note that the join \( P_1 \Join G \) turns some of the input atoms of \( P_1 \) to output atoms. We subtract these atoms from the atoms \( A_1 \) to ensure that \( A_2 \) is a subset of the input atoms of \( P_2 \).

The next function is used to change the truth value of input atoms.

\[ \text{assign}(a, v) : (P, P, (IC, JC), A) \mapsto (P, P, (IC, JC), A_2) \]

for a ground atom \( a \) and a truth value \( v \in \{ t, f \} \) where

- \( A_2 = \begin{cases} A_1 \cup \{a\} & \text{if } v = t \text{ and } a \text{ is an input of } P, \\ A_1 \setminus \{a\} & \text{otherwise.} \end{cases} \)

This function provides the only means to set an input atom to true; the default truth value for input atoms is false. Furthermore, note that calling this function with a non-input atom does not change the system state; their truth values cannot be fixed using this function.

The next function removes the input status from an atom.

\[ \text{release}(a) : (P, P_1, (IC, JC), A) \mapsto (P, P_2, (IC, JC), A_2) \]

for a ground atom \( a \) where

- \( A_2 = A_1 \setminus \{a\} \) and
- \( P_2 = \begin{cases} P_1 \Join (\emptyset, \emptyset, \{a\}) & \text{if } a \text{ is an input of } P_1, \\ P_1 & \text{otherwise.} \end{cases} \)

Again note that the function only affects input atoms, which become output atoms in the resulting system state. This operation permanently sets released atoms to false. The advantage over simply setting the atom to false is that the system can simplify its internal state, completely removing the atom from it.

Finally, operation solve outputs the stable models of the current module filtered by the atoms in \( A \) while leaving the system state unchanged.

\[ \text{solve}() : (P, P, (IC, JC), A) \mapsto (P, P, (IC, JC), A) \]

outputs all stable models \( X \) of \( P \) such that \( A \subseteq X \) and \( X \cap (\text{JI} \setminus A) = \emptyset \) where \( \text{JI} \) are the inputs of \( P \).
$ clingo instance.lp encoding.lp script.lp 0
clingo version 5.5.2
Reading from instance.lp ...
Solving... Solving... Solving... Solving...
Solving... Solving... Solving... Solving...
Solving... Solving... Solving... Solving...
Solving... Solving... Solving... Solving...
Answer: 1
move(1,a,b,1) move(2,a,c,2) move(1,b,c,3) move(3,a,b,4)
move(1,c,a,5) move(2,c,b,6) move(1,a,b,7) move(4,a,c,8)
move(1,b,c,9) move(2,b,a,10) move(1,c,a,11) move(3,b,c,12)
move(1,a,b,13) move(2,a,c,14) move(1,b,c,15)
Models : 1
Calls : 16
Time : 0.10s (Solving: 0.02s 1st Model: 0.00s Unsat: 0.02s)
CPU Time: 0.09s

Listing 3.3.4. Clingo output for incremental ToH example

Note that we can alternatively capture the stable models output by solve by the module
\[ \mathbb{P} \bowtie \mathbb{I} = \{ a \leftarrow \mid a \in A \}, \emptyset, \mathbb{I} \] where \( \mathbb{I} \) are the inputs of \( \mathbb{P} \); the resulting join has no input atoms.

Example 3.3.5. Here we show how the script in Listing 3.3.3 is processed by clingo. The example proceeds similar to Example 3.3.2 by first grounding programs \( P_{\text{base}} \) and \( P_{\text{check}} \) for time step 0 and then grounding programs \( P_{\text{step}} \) and \( P_{\text{check}} \) for time steps \( i > 0 \).

The script defines a main function in Lines 5 to 17. This main function replaces the default behavior of clingo to read, ground, and solve a given program. If a main function is given in a script, the system only reads a program from the given input files and then calls the main function with a Control object corresponding to a system state with a collection of parametrizable extensible programs read from the input files. Remember Example 3.3.4, where we describe how the initial state \( S \) is created by reading the programs in Listings 1.0.1a, 3.3.2, and 3.3.3. Next, we describe how the script proceeds from this state.

In the first iteration of the loop in Lines 7 to 17, \texttt{ground} is called with the list \([\text{"base"}, \emptyset], \text{("check", Number(0))}]\). The Control object changes its state to
\[ S'_0 = \text{ground}(((\text{base}, ()), (\text{check}, (0))))(S). \]

At this point, the input atom \texttt{query(0)} to enable/disable the goal check is false. Before solving, we change its truth value in Line 16 to true and enter the state
\[ S_0 = \text{assign}((\text{query}(0), \text{true}))(S'_0) \]

We then proceed in Line 3.3.2 with computing stable models using the solve operation. The problem is unsatisfiable and the loop enters the next iteration.
At each iteration \( i > 0 \) of the loop, we first release the external atom \( \text{query}(i-1) \) entering state

\[
S_i' = \text{release}(\text{query}(i-1))(S_{i-1}),
\]

and then call \text{ground} with the list \([\text{"step"}, \text{Number}(i)], \text{"check"}, \text{Number}(i)]\). The \text{Control} object changes its state to

\[
S_i = \text{ground}(((\text{step}, (i)), (\text{check}, (i))))(S_i').
\]

Before solving, the current query atom is set to true and we enter state

\[
S_i = \text{assign}(\text{query}(i), t)(S_i').
\]

This process is repeated until the problem becomes satisfiable. Note that our incremental solving mode cannot detect if an instance of our ToH problem is unsatisfiable (for example, with less than three pegs only the top-most disc can be moved). The loop in Lines 7 to 17 never steps for such instances.

We depict the output of the clingo system for our example problem in Listing 3.3.4. Observe that the output is similar to that of the introductory example in Listing 1.0.3. We see additional lines \text{Solving...} for each \text{solve} call and, also, the number of such calls in the short summary at the end. The reported stable model (limited to atoms over predicate \text{move}) corresponds exactly to the one in Listing 1.0.3.

3.3.3 Summary. In Section 2.3, we present \textit{clingo}'s multi-shot solving approach complementing ASP’s declarative input language by procedural control capacities. This is accomplished within a single integrated grounding and solving process in which a program may evolve over time. A program is dynamically extended via grounding. Parts of the grounded program can then be enabled, disabled, or removed via external input atoms. We capture the semantic underpinnings of our approach in terms of module theory and present a state based characterization of the \textit{clingo} API. The procedural control is exercised via traditional programming languages. Currently, the \textit{clingo} API is available for Lua, Python, C, C++ and also for Java, Rust, Prolog, Haskell via external projects. Our framework is geared toward real world applications balancing declarative expressiveness with high performance implementation. To achieve this goal, we cannot arbitrarily extend a program. The evolution of a program is governed by module theory. A system state is captured by a module consisting of a program decorated with its input and output. This module can be extended with further modules given that the modules are compositional. As a result, the high level \textit{clingo} API allows us to address a wide range of problems including unrolling a transition function as in planning [48], interacting with an environment as in robotics [8, 16] or stream reasoning [19], interacting with a user exploring a domain [94], theory solving [104], and advanced forms of search [23]. Our multi-shot solving approach is unique in the field of ASP. While the approach in [33] incrementally grounds a program, it reinitializes the solver at each step.

3.4 Temporal solving

Representing and reasoning about dynamic systems is a key problem in Artificial intelligence and beyond. Accordingly, various formal systems have arisen including calculi
for reasoning about actions and change [128], and temporal logics [57]. In ASP, this is reflected by action languages [97] and recently introduced temporal extensions to the stable model semantics [3]. On the one hand, action languages provide a clean and elegant way to describe transition systems via so-called static and dynamic laws. Although, they are typically translated to ASP, they lack its rich modeling language. Hence, dynamic problems are often modeled directly in ASP by explicitly representing time points and bounding the maximum number of transitions between states of a transition system. The ToH example from the previous section is an example. On the other hand, temporal extensions of the stable model semantics used to deal with models that are infinite sequences of states. This rules out computation by ASP technology and is unnatural to model planning problems where we are only interested in finite sequences of states leading from an initial to some goal state.

In Section 2.5, we define stable models for temporal formulas corresponding to sequences of states—so-called traces. The semantics rests upon a combination of the logic of Here and There [99, HT] and Linear Temporal Logic over finite traces [44, LTLf]. Although, we discuss traces of infinite length in [29], we solely focus on finite traces here. We introduce temporal programs whose semantics is defined via temporal formulas and develop a prototypical system based on multi-shot solving techniques from Section 3.3 to compute their stable models. We illustrate how to model the ToH example from Chapter 1 using temporal programs and solve it using the developed system.

3.4.1 Background. A temporal formula over an alphabet \( \mathcal{A} \) is defined inductively as follows:

- \( a \in \mathcal{A} \) and \( \bot \) are temporal formulas,
- \( \odot F \) is a temporal formula if \( F \) is a temporal formula and \( \odot \in \{ \bullet, \circ \} \),
- \( F \odot G \) is a temporal formula if \( F \) and \( G \) are temporal formulas, and \( \odot \in \{ \lor, \land, \rightarrow, \diamond, \triangleright, \triangleright' \} \).

We refer to \( \bullet, \circ \) and \( \odot \) as previous, trigger and since, and to \( \circ, \triangleright, \triangleright' \) as next, release and until.

We define the derived Boolean connectives \( \neg F = \bot \rightarrow F \) and \( \top = \neg \bot \). Furthermore, we define the following derived temporal operators:

- \( \mathbf{F} = \bot \mathbf{F} \) always before
- \( \circ F = \bot \circ F \) always afterward
- \( \mathbf{F} = \top \mathbf{F} \) eventually before
- \( \circ F = \top \circ F \) eventually afterward
- \( \mathbf{I} = \neg \mathbf{I} \) initial
- \( \mathbf{I} = \neg \mathbf{I} \) final
- \( \mathbf{\mathbf{F}} = \bigcirc (\mathbf{I} \land F) \) initially
- \( \circ F = \bigcirc (\mathbf{I} \land F) \) finally
- \( \mathbf{F} = \mathbf{I} \lor \mathbf{F} \) weak previous
- \( \circ F = \mathbf{I} \lor \mathbf{F} \) weak next

A (finite) trace of length \( n \geq 0 \) over an alphabet \( \mathcal{A} \) is a sequence \((X_i)_{0 \leq i \leq n}\) of sets such that each \( X_i \subseteq \mathcal{A} \). In the following, we are interested in temporal stable models of formulas, which are traces that satisfy a set of temporal formulas as well as a minimality/derivability criteria formally defined in Section 2.5.

Similar to the previous sections, we introduce a simplified version of the material in Section 2.5 to present the main ideas and concepts. Thus, we consider a simple class of temporal formulas that have a straight-forward mapping to the incremental programs introduced in Section 3.3.
3.4. TEMPORAL SOLVING

Let \( \Sigma \) be a signature. A temporal atom over \( \Sigma \) has form \( a, \bullet a, \) or \( \bullet \hat{a} \) where \( a \) is an atom over \( \Sigma \). A temporal literal over \( \Sigma \) is either formula \( \overline{a} \), or has form \( a \) or \( \neg a \) for a temporal atom \( a \) over \( \Sigma \). A temporal program over \( \Sigma \) consists of temporal rules over \( \Sigma \) of form

\[ h \leftarrow B \]
\[ \delta a(h \leftarrow B) \]
\[ a(h \leftarrow B) \]

where \( h \) is an atom over \( \Sigma \) and \( B \) is a set of temporal literals over \( \Sigma \). Rules of form (11), (12), and (13) are called initial, dynamic, and always rules, respectively.

We define instances of temporal rules in analogy to Section 3.1 for programs. An instance of a temporal rule is obtained by substituting constants from its signature for all its variables. We use \( \Gamma(P) \) to denote the set of all instances of rules in a temporal program \( P \).

The semantics of temporal programs is given via a translation to temporal formulas. We translate a temporal program \( P \) into a temporal formula over the atom base of \( P \) using translation \( \phi \) as follows:

\[ \phi(h \leftarrow B) = \bigwedge_{l \in B} l \rightarrow h, \]
\[ \phi(\overline{a}) = a \phi(r), \]
\[ \phi(\delta \overline{a}) = \delta a \phi(r), \] and
\[ \phi(P) = \{ \phi(r) \mid r \in \Gamma(P) \}. \]

As in Section 2.5, the temporal stable models of temporal program \( P \) are the temporal stable models of \( \phi(P) \).

Observe that initial rules of form (11) over standard literals are syntactically equivalent to standard rules. In fact, temporal stable models of length zero of a set of such rules directly correspond to their standard stable models. Thus, we can see standard programs as temporal programs that consist of initial rules only. Dynamic rules of form (12) apply to all but the initial time step. Such rules can be used to specify how to transition from one time step to the next. For example, in our ToH example, we can encode the inertia of discs as follows:

\[ \delta \overline{a}\overline{o}(D,P) \leftarrow a\overline{o}(D,P), \neg no(D,P) \].

Always rules of form (13) not just apply to the initial time point but to all time points. We can express for example that solutions of our ToH problem have to be consistent:

\[ a(f \leftarrow \overline{f}, \overline{o}(D,P), no(D,P)). \]

Note that an always rule \( a r \) can equivalently be written using the initial and dynamic rules \( r \) and \( \delta r \).

To compute temporal stable models, we use multi-shot solving as introduced in Section 3.3. We translate a temporal program \( P \) over a signature \( \Sigma \) into a collection of parametrizable programs over signature \( \Sigma' \). The signature \( \Sigma' \) is obtained from \( \Sigma \) by incrementing the arities of predicates in \( \Sigma \) by one. Furthermore, we add a fresh
3.4. TEMPORAL SOLVING

predictate $f$ with arity 1 and fresh constants $t$ and $t'$ for parameters. We define the translation $\tau$ as follows:

$$
\tau(p(t)) = p(t, t),
$$

$$
\tau(\bullet p(t)) = p(t, t'),
$$

$$
\tau(\bullet p(t)) = p(t, 0),
$$

$$
\tau(\neg a) = \neg \tau(a),
$$

$$
\tau(h \leftarrow B) = \tau(h) \leftarrow \{ \tau(l) \mid l \in B \},
$$

$$
\tau(P) = \langle P \rangle_{n \in \{\text{init, dynamic, always} \}}
$$

where

$$
P_{\text{init}} = \{ \tau(r) \mid r \in P \},
$$

$$
P_{\text{dynamic}} = \{ \tau(r) \mid \boxdot r \in P \},
$$

$$
P_{\text{always}} = \{ \tau(r) \mid r \in P \} \cup \{ f(t) \leftarrow \epsilon \}.
$$

are parametrizable extensible programs with parameters $t$ and $t'$.

Next, we use the translation $\tau$ and the state based operations from Section 3.3 to obtain states capturing the temporal stable models of a temporal program. We inductively define states in such a way that a state $S_i$ captures all temporal stable models of length $i \geq 0$ of a temporal program. Given a temporal program $P$, we let

$$
S_0'' = (\phi(P), (\emptyset, \emptyset, \emptyset), (\emptyset, \emptyset, \emptyset))
$$

$$
S_0' = \text{ground}(((\text{init}, (-1, 0)), (\text{always}, (-1, 0))))(S_0''),
$$

$$
S_0 = \text{assign}(f(0), t)(S_0')
$$

and for $i > 0$

$$
S_i'' = \text{release}(f(i - 1))(S_{i-1}),
$$

$$
S_i' = \text{ground}(((\text{always}, (i - 1, i)), (\text{dynamic}, (i - 1, i))))(S_i''),
$$

$$
S_i = \text{assign}(f(i), t)(S_i').
$$

Then, the current and grounded module when obtaining states $S_i$ are compositional, and there is a one-to-one correspondence between the temporal stable models of length $n$ of temporal program $P$ and the stable models output by $\text{solve}(S_n)$. That is, a temporal stable model $(X_i)_{0 \leq i \leq n}$ corresponds to the stable model $\{ p(t, i) \mid p(t) \in X_i, 0 \leq i \leq n \} \cup \{ f(n) \}$ as output by $\text{solve}$.

**Example 3.4.1.** We show how to use temporal programs to obtain temporal stable models specifying solutions of our simplified ToH example.

Since initial rules are syntactically equivalent to standard rules, we use the rules (without time parameter) in program $P_{\text{base}} \setminus \{r_3\}$ from Example 3.3.2 as initial rules. From rule $r_3$, we remove the time parameter and use rule $r_3''$ in Listing 3.4.1. We next turn to the dynamic rules $r_4''$, $r_5''$, $r_6''$, and $r_7''$ in Listing 3.4.1. These rules correspond to rules $r_4'$, $r_5'$, $r_6'$, and $r_7'$ from program $P_{\text{step}}$ in Example 3.3.2. As compared to the
3.4. TEMPORAL SOLVING

We now develop a simplified version of the telingo system that supports temporal programs as input. To represent temporal programs, we use `#program` directives to partition rules in initial, dynamic and always rules. For convenience, we also treat rules in the default base program as initial rules. This allows us to use problem instances across clingo and telingo. For example, we can use the ToH instance from Listing 1.0.1a without change in telingo. The • and ◆ operators are represented by preceding an atom with a prime and an underscore, respectively. To encode the ◬ operator, we use clingo’s theory language (in a limited form), which allows us to precede an identifier with an ampersand.

Example 3.4.2. Similar to Example 3.3.4, we forgo extended language constructs not supported by our simple temporal programs and give an encoding for the ToH problem in telingo syntax in Listing 3.4.2.

Unlike in Example 3.4.1, we have to resort to the available clingo syntax to represent temporal operators. Initial rules are given in Lines 1 to 3, always rules in Lines 5 to 11, and dynamic rules in Lines 13 to 27. To encode the temporal operators • and ◆, we change the semantics of some of clingo’s identifiers. For example, in Line 15, we write `_on(D,P)` instead of `on(D,P)` and, in Line 7, we write `_on(D,P)` instead of `on(D,P)`. In Line 11, we write `&final` instead of ◬ making use of clingo’s theory language extension. Also note that we support `#show` statements to limit the output as used in Line 30.

In the following, we draw on material from Section 2.4 using clingo’s API to implement a simplified version of the telingo system. In particular, we use clingo’s abstract syntax tree (AST) to transform a temporal program into an incremental one. We rewrite the standard rules, we omit time parameters in these rules applying operator • to literals referring to the previous time point and operator ◆ to literals referring to the initial time point. In the same way, we obtain the always rules `r'8` and `r'9` from rules `r'8` and `r'9` in program Palways.

Furthermore, observe that the states $S_i$ to capture temporal stable models are defined similarly as in Example 3.3.5. In fact, the only difference is that (for technical reasons) initial rules additionally receive a time parameter in the translation, the parameter names differ, and that atom query($i$) is called $f(i)$. Thus, it is easy to see that states for the same length $i$ have corresponding stable models for our example programs.
3.4. TEMPORAL SOLVING

### Listing 3.4.2. Temporal ToH encoding

```
#program initial.
% establish initial situation
on(D,P) :- init(D,P).

#program always.
% uniqueness of location: a disc can only be on one peg
-on(D,Q) :- on(D,P), _peg(Q), P!=Q.

% ensure that the goal was reached
:- &final, _goal(D,P), not on(D,P).

#program dynamic.
% choose discs to move
{ move(D,P,Q) } :- 'on(D,P), _peg(Q), P!=Q.

% there must be at most one move per time step.
:- #count { D,P,Q; move(D,P,Q) } > 1.
% only the topmost disc can be moved
:- move(D,P,_), 'on(E,P), D>E.
% a disc can only be put on larger discs
:- move(D,_,Q), 'on(E,Q), D>E.

% effects: change the location of the moved disc
on(D,Q) :- move(D,_,Q).
% inertia: discs stay in place by default
on(D,P) :- 'on(D,P), not -on(D,P).

#show move/3.
```

AST using the **Transformer** class, which implements the visitor pattern to transform nodes in the AST of a program. Since we have to rewrite the input program, we cannot use an embedded script as in Section 3.3 and rather use **clingo’s Application** class to customize how a program is parsed, grounded, solved and its solutions are output. We split the implementation of our simplified **telingo** system into three parts: a transformer to handle temporal atoms, a transformer to handle temporal literals and temporal rules, and an application class to implement parsing, grounding, solving and output.

We proceed bottom up and begin by implementing translation $\tau$ for temporal atoms in Listing 3.4.3. The **AtomTransformer** in the listing derives from **clingo’s Transformer** class, which implements a visitor that allows us to change the AST of a **clingo** program. The transformer recursively visits each node of an AST. We implement function **visit Function** in Lines 18 to 22, which is called whenever the transformer visits a node.
from clingo.symbol import Function, Number
import clingo.ast as ast

class AtomTransformer(ast.Transformer):
    def _prm(self, fun, loc):
        if fun.startswith('_'):
            return fun[1:], ast.SymbolicTerm(loc, Number(0))
        if fun.startswith('"'):
            prm = ast.BinaryOperation(
                loc, ast.BinaryOperator.Minus,
                ast.SymbolicTerm(loc, Function('_t')),
                ast.SymbolicTerm(loc, Number(1)))
            return fun[1:], prm
        return fun, ast.SymbolicTerm(loc, Function('_t'))

    def visit_Function(self, trm):
        fun, prm = self._prm(trm.name, trm.location)
        trm = trm.update(name=fun)
        trm.arguments.append(prm)
        return trm

Listing 3.4.3. Transformer to rewrite atoms

of type function. Note that atoms in clingo share the same syntax as terms, thus, they are represented using function terms. Our transformer only visits the outermost function term, transforms it, and then stops further processing. Remember that translation $\tau$ for temporal atoms simply appends a time parameter to the arguments of an atom. Helper function _prm determines the time parameter and strips operators encoded in the function name. The result of this function is then used to update the function name and its parameters in Lines 20 and 21, respectively. Our helper function proceeds as follows. In Lines 6 to 7, we handle functions starting with an underscore representing operator $\hat{\cdot}$ and return the function name stripped from its initial underscore together with term $0$ to refer to the initial time step. For example, given function name "on", the function returns a pair consisting of string "on" and term 0. In Lines 9 to 14, we proceed similarly but return the term $\_t-1$ to refer to the previous time step. Note that we precede the parameter with two underscores; this a convention to avoid conflicts with other identifiers used in encodings. For example, given function name "'on", the function returns a pair consisting of string "on" and term $\_t-1$. In Line 9, we simply return term $\_t$ to refer to the current time step. For example, given function name "on", the function returns a pair consisting of string "on" and term $\_t$.

We next turn to the ProgramTransformer class in Listing 3.4.4. This class is responsible to translate temporal atoms, final operators, #program directives, and #show statements. Function visit_SymbolicAtom in Lines 31 to 32 is used to translate temporal
3.4. TEMPORAL SOLVING

import clingo.ast as ast

class ProgramTransformer(ast.Transformer):
    def __init__(self):
        self.atf = AtomTransformer()

    def visit_SymbolicAtom(self, atm):
        return atm.update(symbol=self.atf(atm.symbol))

    def visit_TheoryAtom(self, atm):
        if atm.term.name == 'final':
            loc = atm.location
            prm = ast.SymbolicTerm(loc, Function('_t'))
            fun = ast.Function(loc, '__final', [prm], False)
            return ast.SymbolicAtom(fun);
        return atm

    def visit_Program(self, prg):
        nms = ('base', 'initial', 'always', 'dynamic')
        if prg.name in nms:
            pms = [ast.Id(prg.location, '_t')]
            return prg.update(parameters=pms)
        return prg

    def visit_ShowSignature(self, sig):
        return sig.update(arity=sig.arity + 1)

Listing 3.4.4. Transformer to rewrite programs

atoms using the AtomTransformer from Listing 3.4.3; it simply adds a time parameter to the atom. For example, 'on(D,P) representing the temporal atom on(D,P) is translated to on(D,P,_t). Function visit_TheoryAtom in Lines 34 to 40 replaces the theory atom __final to represent F by the ordinary atom __final(t). We again use two underscores to mark this atom for internal use. Function visit_Program in Lines 42 to 47 adds a time parameter to the program directives base, initial, always, and dynamic to gather the respective temporal rule types. Note that we also consider the base program here, which is treated the same way as the initial program. For example, the directive #program dynamic is replaced by #program dynamic(_t). Finally, in Lines 49 to 50, we implement visit_ShowSignature to adjust #show statements by incrementing their arity by one. This is necessary because the arity of all predicates increased by one by adding a time parameter. With this, we can reuse clingo’s output functionality to only show selected atoms. For example, the directive #show move/3 is replaced by #show move/4.

Our simplified telingo system is implemented in Listing 3.4.5 using the classes from Listings 3.4.3 and 3.4.4. We define the TelApp class deriving from clingo’s Application
from clingo.application import clingo_main, Application
from clingo.symbol import Function, Number, SymbolType
import clingo.ast as ast
from sys import exit, stdout

class TelApp(Application):
    def print_model(self, mdl, prt):
        tab = {}
        for s in mdl.symbols(shown=True):
            if s.type == SymbolType.Function and s.arguments:
                sms = tab.setdefault(s.arguments[-1], [])
                sms.append(Function(s.name, s.arguments[:-1]))
            for stp, sms in sorted(tab.items()):
                stdout.write(f" State {stp}:")
        sig = None
        for sym in sorted(sms):
            if (sym.name, len(sym.arguments)) != sig:
                stdout.write("\n")
                sig = (sym.name, len(sym.arguments))
                stdout.write(f" {sym}")
                stdout.write("\n")
        def main(self, ctl, files):
            with ast.ProgramBuilder(ctl) as bld:
                ptf = ProgramTransformer()
                ast.parse_files(files, lambda s: bld.add(ptf(s)))
                ctl.add('always', ['_t'], '#external __final(_t).')
                stp, ret, fin = 0, None, None
                while ret is None or not ret.satisfiable:
                    pts = [('always', [Number(stp)])]
                    if fin is not None:
                        ctl.release_external(fin)
                        pts.append(('dynamic', [Number(stp)]))
                    else:
                        pts.append(('base', [Number(stp)]))
                    ctl.ground(pts)
                    fin = Function('__final', [Number(stp)])
                    ctl.assign_external(fin, True)
                    ret, stp = ctl.solve(), stp + 1
            exit(clingo_main(TelApp()))

Listing 3.4.5. Application to solve temporal programs
Listing 3.4.6. Telingo output for temporal ToH example

class. An instance of this class is then used in Line 94 to launch the telingo system. Apart from functionality overridden in the derived class, the system behaves as clingo. We now turn to the overridden functionality. By overriding function main, we change the way a program is parsed, grounded, and solved. In Lines 76 to 78, we first parse the files passed to the telingo system and then apply the ProgramTransformer developed in Listing 3.4.4 to each parsed statement captured by an AST. The remainder of the function proceeds in a similar fashion as Listing 3.3.3 in Section 3.3. Remember that we introduced predicate __final to capture operator F. In Line 79, we add a program to declare corresponding atoms as external. This corresponds to the external query atom from Listing 3.3.3. The only difference is that we add it via clingo’s API. The code in Lines 80 to 92 corresponds to the main function in Listing 3.3.3. Only the programs to ground change. Program base is now called initial (or base) and called with parameter __t=0, program step is called dynamic, and program check is called always. Finally, function print_model, in Lines 59 to 73, changes how a model is printed. The function outputs a stable model of the rewritten program as a temporal stable model by listing atoms state by state.

Example 3.4.3. In Listing 3.4.6, we show the output of running the script in Listings 3.4.3 to 3.4.5 run with the encodings in Listings 1.0.1a and 3.4.2. Observe that the output closely resembles the one in Listing 3.4.3 of Example 3.3.5. The only difference is how the model is printed. In fact, the output would be equal if we had not overridden function print_model.
3.5. THEORY SOLVING

3.4.3 Summary. We have sketched the design and implementation of an ASP system extended by constructs from Linear Temporal Logic. The logical foundations of this approach were laid in Section 2.5, where we introduce Temporal Equilibrium Logic for finite traces, TEL$_f$, based on the logic of Here and There and Linear Temporal Logic for finite traces. In Section 2.5, we prove that there is a simple normal form for TEL$_f$ motivating our definition of temporal programs. We have drawn on these foundations by designing a corresponding extension of the ASP system clingo. The described prototypical implementation reflects the one of the actual telingo system. Moreover, it illustrates the great practical value of clingo’s AST that allows for clean and easy extensions of ASP systems and, also, how multi-shot solving can be used as a solving infrastructure for sophisticated forms of reasoning such as temporal reasoning.

From a modeling perspective, temporal programs have several advantages as compared to the incremental solving mode introduced in Section 3.3. This can be seen comparing the encodings for the ToH problem presented in this and the previous section. For one, the encoding is more declarative because we do not have to explicitly model time points using integers. For another, we only permit operators referring to the past in rule bodies. This ensures that temporal programs can be translated to modular incremental programs. In Section 2.5, we further extend this class to so called present-centered temporal programs. This class permits to use temporal operators referring to the future in rule heads and the past in rule bodies, and arbitrary temporal operators in constraints.

In conclusion, we obtain a system offering an expressive and semantically well founded language for modeling dynamic systems in ASP implemented using existing solving technology.

3.5 Theory solving

Answer set programming has become an established paradigm to solve combinatorial search problems. Despite its versatility, it sometimes falls short in handling non-Boolean domains. In particular, this applies to problems involving constraints over large integer domains. An example class for such problems are scheduling problems [39] where tasks have to be assigned to time points meeting a set of constraints, like limited processing resources and deadlines to finish tasks. The problem here is that explicitly modeling time points in ASP results in large ground programs and more compact representations, like binary encodings for integers, harm the search efficiency. In Section 2.4, we address this shortcoming by providing the means to implement generic theory reasoning. This spans from theory grammars to extend the input language to theory propagators to extend the search itself. We have already seen how to extend clingo’s input language in Section 3.4.

In this section, we show how to extend clingo’s search to propagate difference constraints over integer variables and illustrate how such constraints can be used to encode the flow shop scheduling problem [137].

Section 2.6 comprises the work in [104], presenting three different systems to extend clingo with constraints over integers and even floating point numbers, namely, clingo[lp], clingo[dl], and clingcon. The former supports linear constraints over floating point numbers relying on external LP solvers [43], while the latter two systems implement specialized algorithms for integer constraints tightly integrated into clingo’s search. However, all three systems build upon clingo’s infrastructure for theory propagation. In
we develop a simplified version of the clingo\cite{dl} system supporting rules with difference constraints in their heads. Even though the system only offers a restricted form of constraints, it is particularly well suited for solving scheduling programs.

3.5.1 Background. We begin by extending the programs introduced in Section 3.1 with difference constraints. A dl-atom has form

\[(a - b) \leq c \tag{14}\]

where \(a\) and \(b\) are term tuples, and \(c\) is a term optionally preceded by a minus sign. A dl-program is a set of rules of form

\[h \leftarrow B \tag{15}\]

where \(h\) is an atom or dl-atom and \(B\) is a set of body literals.

We say that a ground dl-atom of form (14) is well-formed if \(c\) is an integer; ground atoms are well-formed. An instance of a rule is obtained by replacing all variables in it by constants such that its head is well-formed. The grounding \(\Gamma(P)\) for a dl-program \(P\) is defined as in Section 3.1 using the above instance definition. Further notions, like interpretations and stable models defined for programs in Section 3.1, are extended to dl-programs by treating ground well-formed dl-atoms like atoms.

We now define the semantics of dl-programs. An interpretation is dl-consistent if there is a mapping \(\lambda\) from tuples of constants to integers such that for all dl-atoms of form (14) in the interpretation, we have \(\lambda(a) - \lambda(b) \leq c\). Given a dl-program \(P\), an interpretation is a dl-stable model of \(P\) if it is a dl-consistent stable model of \(P\).

Example 3.5.1. In the following, we consider the permutation flow shop problem about scheduling the tasks \(t_1, \ldots, t_n\) on processing units \(u_1, \ldots, u_m\) for \(n, m > 0\) in such a way that the tasks are processed within a given time budget \(b\). Each task \(t_i\) takes time \(d_{i,j}\) to be processed on a processing unit \(u_j\). A schedule \(\lambda\) mapping a task and machine pair to a positive integer is subject to the following conditions:

1. \(\lambda(t, u) \leq \lambda(t', u)\) implies \(\lambda(t, u') \leq \lambda(t', u')\) for all tasks \(t\) and \(t'\), and units \(u\) and \(u'\), that is, the tasks are processed in the same order on all machines,
2. \(\lambda(t_i, u_k) + d_{i,k} \leq \lambda(t_j, u_k)\) or \(\lambda(t_j, u_k) + d_{j,k} \leq \lambda(t_i, u_k)\) for all units \(u_k\), and tasks \(t_i\) and \(t_j\) with \(i \neq j\), that is, no two tasks can run at the same time on a unit, and
3. \(\lambda(t_i, u_j) + d_{i,j} \leq \lambda(t_i, u_k)\) for all tasks \(t_i\), and units \(u_j\) and \(u_k\) with \(j < k\), that is, a task has to be processed on a unit before going to the next one,
4. \(\lambda(t_i, u_j) + d_{i,j} \leq b\) for all tasks \(t_i\) and units \(u_j\), that is, all tasks have to finish within the given budget.

We use the instance depicted in Figure 3.5.1 with \(n = 3\) tasks and \(m = 2\) units. As given in the first column, we have the tasks \(t_1\), \(t_2\), and \(t_3\). The processing times of a task are given by the colored squares where lighter shades capture the duration on unit \(u_1\) and darker shades the duration on unit \(u_2\). We have \(d_{1,1} = 3\), \(d_{2,1} = 1\), \(d_{3,1} = 5\), \(d_{1,2} = 4\), \(d_{2,2} = 6\), and \(d_{3,2} = 5\).
We represent this problem using the following facts:

\[
\begin{align*}
\text{b}(16) & \quad t(1) & \quad t(2) & \quad t(3) \\
u(1) & \quad d(1, 1, 3) & \quad d(2, 1, 1) & \quad d(3, 1, 5) \\
u(2) & \quad d(1, 2, 4) & \quad d(2, 2, 6) & \quad d(3, 2, 5) \\
\text{ord}(1, 2) & \quad \text{ord}(2, 3)
\end{align*}
\]

We use atoms \(t(i), u(j), d(i, j, d_{ij}), \) and \(b(b)\) to capture tasks \(t_i\), units \(u_j\), durations \(d_{ij}\), and budget \(b\), respectively. Furthermore, we use \(\text{ord}(i, i + 1)\) for \(1 \leq i \leq \max\{n, l\}\) to order task and unit indices in ascending order.

To model this problem with a dl-program, we guess an order in which to process tasks. A task is scheduled on a processing unit before succeeding tasks are scheduled on the same unit. In Figure 3.5.2, we depict the two solutions of this form for our example instance in Figure 3.5.1. In the first solution, we first process task \(t_2\), then task \(t_1\), and task \(t_3\) at the end. This takes exactly 16 time units, which is the maximum our budget permits. The solution can be verified in the picture by counting the number of squares in the picture from left to right. The second solution has the same cost and just exchanges the order in which the last two tasks from the previous solution are processed.
Next, we encode the problem as a dl-program:

\[
\begin{align*}
(r_1) & \quad a(S, T) \leftarrow \neg n a(S, T), t(T), t(S) \\
(r_2) & \quad n a(S, T) \leftarrow \neg a(S, T), t(T), t(S) \\
(r_3) & \quad \text{assigned}(S) \leftarrow a(S, T) \\
(r_4) & \quad f \leftarrow \neg f, t(S), \neg \text{assigned}(S) \\
(r_5) & \quad eq(T, T) \leftarrow \neg t(T) \\
(r_6) & \quad f \leftarrow \neg f, a(S, T), a(S, T'), \neg eq(T, T') \\
(r_7) & \quad (S, U) - (S', U) \leq -D \leftarrow a(S, T), t(S'), \text{ord}(S, S'), d(T, U, D) \\
(r_8) & \quad (S, U) - (S, U') \leq -D \leftarrow a(S, T), u(U'), \text{ord}(U, U'), d(T, U, D) \\
(r_9) & \quad (s) - (S, U) \leq 0 \leftarrow t(S), u(U) \\
(r_{10}) & \quad (S, U) - (e) \leq -D \leftarrow t(S), u(U) \\
(r_{11}) & \quad (e) - (s) \leq B \leftarrow b(B)
\end{align*}
\]

We use atom \( a(s, i) \) to assign tasks \( t_i \) to slots \( 1 \leq s \leq n \); a task assigned to a slot \( s \) is processed before a task assigned to slot \( s' \) if \( s < s' \). The assignment is encoded in rules \( r_1 \) to \( r_6 \). Rules \( r_1 \) and \( r_2 \) guess an assignment of tasks to slots, rules \( r_3 \) and \( r_4 \) ensure that at least one task is assigned to a slot, and rules \( r_5 \) and \( r_6 \) ensure that at most one task is assigned to a slot. This assignment is then used in the following rules to derive dl-atoms that ensure properties (1) to (4) for schedules. We use tuples \( (s, i) \) to capture the starting time of a task assigned to slot \( s \) on unit \( u_i \). With this, rule \( r_7 \) encodes properties (1) and (2), and rule \( r_8 \) encodes property (3). Furthermore, we use tuples \( (s) \) and \( (e) \) to capture start and end times, which according to rule \( r_{11} \) must be no more than \( B \) time units apart. Rules \( r_9 \) and \( r_{10} \) encode that all starting and finishing times of tasks on units must be greater or equal to the start and less or equal to the finishing times, respectively. Thus, rules \( r_9 \) to \( r_{11} \) ensure property (4).

For our example instance, this encoding has the two dl-stable models:

\[
\{a(1, 2), a(2, 3), a(3, 1)\} \text{ and } \{a(1, 2), a(2, 1), a(3, 3)\},
\]

restricted to atoms over predicate \( a \), corresponding to the solutions in Figure 3.5.2.

### 3.5.2 Theory specification

We now show how to use clingo’s input language to specify difference constraints. The input language provides syntax for theory specifications to define theory atoms that can be used in rules. A theory specification consists of a list of theory term and theory atom definitions. The former specify which kind of terms can be used in theory atoms. The latter specify which kind of theory atoms can be used in rules. For a formal description, we refer to Section 2.4. Here, we only explain the theory specification in Listing 3.5.1 to extend a program with difference constraints in rule heads.

The code in Lines 2 to 10 is an example for a theory term definition. It has name term and contains a set of operator definitions within braces. Each operator is associated with a priority, an operator type, and an associativity in case of a binary operator. An operator can be either binary or unary. Binary operators are either left or right
associative. Using this theory term definition, we can construct theory terms like

\[(S, U) - (s, U+1)\text{ or } S + U + 2*D.\]

The precedence and associativity of operators ensures that the second term uniquely corresponds to the term \((S+U)+(2*D).\) Furthermore, note that the first term can be used on the left hand side of a difference constraint while the second cannot. A theory term definition only describes the operators that can be used to construct theory terms, additional checks have to be performed programmatically using the clingo API.

The code in Line 11 is an example for a theory atom definition. It allows us to construct theory atoms with atoms over predicate \textit{diff} with arity 0 as name using theory terms specified by theory term definition \textit{term} as elements. Furthermore, we can use <= and a theory term specified by theory term definition \textit{term} as guard. Finally, the theory atom is restricted to occur in rule heads via label \textit{head}. Using this theory atom definition, we can construct theory atoms like

\[
\textit{\&diff} \{ (S, U)-(s, U+1) \} <= -D \text{ or } \\
\textit{\&diff} \{ S+U+2*D \} <= -D.
\]

As above, invalid difference constraints like the second one have to be discarded programmatically.

**Example 3.5.2.** We now have the necessary syntax to compactly encode the flow shop problem from Example 3.5.1 in Listings 3.5.2 and 3.5.3 using clingo’s rich input language.
In Lines 1 to 2, we encode the assignment from tasks to slots using aggregates. The rules in Line 4 and 5 correspond to rules $r_7$ and $r_8$ from Example 3.5.1, respectively. The rules in Lines 6 to 7 correspond to rules $r_9$ to $r_{11}$. Note that the term 0 in the difference constraint is used like tuple (s) in rules $r_9$ and $r_{11}$. In fact, clingo\[DL\] treats this value specially when outputting mappings witnessing dl-consistency. We delay further details until we have described the output of our simplified clingo\[DL\] implementation for this example. Furthermore, we do not need to introduce tuple (e) because we can subtract the duration from the bound in Line 7 combining rules $r_{10}$ and $r_{11}$.

When we pass the programs in Listings 3.5.2 and 3.5.3 along with the theory from Listing 3.5.1 to clingo, we obtain 6 stable models—one for each assignment of tasks to slots. In the following, we show how to use the API to implement a propagator to discard stable models that are not dl-consistent.

3.5.3 Theory solving. The algorithmic approach to ASP solving modulo theories of clingo is based on Conflict-Driven Constraint Learning (CDCL). From a high-level point of view, clingo’s solving component translates a ground program into a set of nogoods or constraints implicitly capturing a set of nogoods \[79\]. The solving component then tries to find assignments that satisfy this set of nogoods. Nogoods and assignments are sets of literals where a set of nogoods is satisfied by an assignment if none of them is a subset of the assignment. Each satisfying assignment corresponds to a stable model of the original program.

To implement a particular theory, the clingo API offers the propagator interface that can be implemented by a custom propagator to implicitly represent a set of nogoods. In our difference constraint propagator, a nogood consists of literals referring to theory atoms that cannot be satisfied at the same time. The propagator implicitly captures the set of all such nogoods and informs the solver at key points about them. These key points are determined by the backtracking-based search for satisfying assignments. The search proceeds by incrementally extending an assignment via unit propagation and backtracking as soon as it determines that an assignment can no longer be extended to a valid solution (that is, as soon as a nogood conflicts with the current assignment). The propagator interface provides two methods to customize this process. Method propagate is called whenever an assignment is extended with literals relevant to the propagator. In our simple difference constraint propagator, we only use this function to inform the
solver about nogoods that became conflicting, forcing the solver to backtrack. It is also possible to inform the solver about unit resulting nogoods, which we do not discuss here; further information can be found in [40]. Method undo is called whenever the solver backtracks. For performance reasons, we incrementally update the state of the propagator during propagation. Here, we simply reset the state taking back changes done during propagation until a point is reached from which the search can continue. For details, we refer to Section 2.6.

Our example propagator for difference constraints in Listing 3.5.4 implements the algorithm presented in [40]. The idea is that deciding whether a set of difference constraints is satisfiable can be mapped to a graph problem. Given a set of difference constraints, let \((V, E)\) be the weighted directed graph such that \(V\) is the set of variables occurring in the constraints and \(E\) the set of weighted edges \((u, v, d)\) for each constraint \(u - v \leq d\). The set of difference constraints is satisfiable if the corresponding graph does not contain a negative cycle (that is, a cycle whose sum of edge labels is negative). The Graph class is responsible for cycle detection; we refrain from giving its code and rather concentrate on describing its interface:

- Function \texttt{add\_edge} adds an edge of form \((u, v, d)\) to the graph. If adding an edge to the graph leads to a negative cycle, the function returns the cycle in form of a list of edges; otherwise, it returns \texttt{None}. Furthermore, each edge added to the graph is associated with a decision level. This additional information is used to backtrack to a previous state of the graph, whenever the solver has to backtrack to recover from a conflict.
- Function \texttt{backtrack} takes a decision level as argument. It removes all edges added on that level from the graph.
- The class internally maintains a mapping from nodes to integers. This mapping can be turned into a mapping from terms to integers witnessing dl-consistency. Function \texttt{get\_assignment} returns this mapping in form of a list of pairs of terms and integers.

The difference constraint propagator implements the Propagator interface in Listing 3.5.4 in Lines 113 to 157; it features aspects like incremental propagation and backtracking, while supporting solving with multiple threads, and multi-shot solving. Whenever the set of edges associated with the current partial assignment of a solver induces a negative cycle and, hence, the corresponding difference constraints are unsatisfiable, it adds a nogood forbidding the negative cycle. To this end, it maintains data structures for detecting whether there is a conflict upon addition of new edges. More precisely, the propagator has three data members:

1. The \_12e dictionary in Line 115 maps solver literals for difference constraint theory atoms to their corresponding edges,
2. the \_e2l dictionary in Line 116 maps edges back to solver literals, and
3. the \_states list in Line 117 stores for each solver thread its current graph with the edges assigned so far.

Function \texttt{init} in Lines 119 to 128 sets up watches as well as the dictionaries \_12e and \_e2l. To this end, it traverses the theory atoms over predicate \&diff in Lines 120 to 128. Note that the loop simply ignores other theory atoms treated by other propagators. In Lines 122 to 124, we extract the edge from the theory atom. Each such atom is associated
class DLPropagator(Propagator):
    def __init__(self):
        self._l2e = {}
        self._e2l = {}
        self._states = []

    def init(self, init):
        for a in init.theory_atoms:
            if a.term.name == "diff" and not a.term.arguments:
                u = _eval(a.elements[0].terms[0].arguments[0])
                v = _eval(a.elements[0].terms[0].arguments[1])
                w = _eval(a.guard[1]).number
                l = init.solver_literal(a.literal)
                self._l2e.setdefault(l, []).append((u, v, w))
                self._e2l.setdefault((u, v, w), []).append(l)
            init.add_watch(l)

    def _state(self, thread_id):
        while len(self._states) <= thread_id:
            self._states.append(Graph())
        return self._states[thread_id]

    def _lit(self, ctl, edge):
        for lit in self._e2l[edge]:
            if ctl.assignment.is_true(lit):
                return lit

    def propagate(self, ctl, changes):
        state = self._state(ctl.thread_id)
        level = ctl.assignment.decision_level
        for lit in changes:
            for edge in self._l2e[lit]:
                cycle = state.add_edge(level, edge)
                if cycle is not None:
                    c = [self._lit(ctl, e) for e in cycle]
                    ctl.add_nogood(c) and ctl.propagate()
                    return

    def undo(self, thread_id, amt, changes):
        self._state(thread_id).backtrack(amt.decision_level)

    def on_model(self, model):
        amt = self._state(model.thread_id).get_assignment()
        model.extend([Function("dl", [var, Number(value)])
                      for var, value in amt])

Listing 3.5.4. Propagator for difference constraints
class DLApp(Application):
    program_name = "clingo-dl"
    version = "1.0"

def __init__(self):
    self._propagator = DLPropagator()

def on_model(self, model):
    self._propagator.on_model(model)

def main(self, ctl, files):
    ctl.register_propagator(self._propagator)
    ctl.load("dl-theory.lp")
    for path in files:
        ctl.load(path)
    if not files:
        ctl.load("-")
    ctl.ground(["base", []])
    ctl.solve(on_model=self.on_model)
    sys.exit(int(clingo_main(DLApp(), sys.argv[1:1])))

Listing 3.5.5. Application to solve dl-programs

with a solver literal, obtained in Line 125. The mappings between solver literals and corresponding edges are then stored in the _l2e and _e2l dictionaries in Lines 126 and 127. In Line 128 of the loop, a watch is added for each solver literal at hand, so that the solver calls function propagate whenever the edge has to be added to the graph.

Function propagate, given in Lines 140 to 149, accesses ctl.thread_id in Line 141 to obtain the graph associated with the active thread. The loops in Lines 143 to 149 then iterate over the list of changes and associated edges. In Line 145, each such edge is added to the graph. If adding the edge produces a negative cycle, a nogood is added in Line 148. Because an edge can be associated with multiple solver literals, we use function _lit retrieving the first solver literal associated with an edge that is true, to construct the nogood forbidding the cycle. Given that the solver has to resolve the conflict and backtrack, the call to add_nogood always returns false, so that propagation is stopped without processing the remaining changes any further.

Given that each edge added to the graph in Line 3.5.3 is associated with the current decision level, the implementation of function undo in Lines 151 to 152 is straight-forward. It calls function backtrack on the solver thread’s graph to remove all edges added on the current decision level.
Listing 3.5.6. \texttt{Clingo[dl]} output for flow shop example

Listing 3.5.5 shows the application that registers our propagator and addresses grounding and solving. Lines 170 to 180 implement a customized main function. The difference to \texttt{clingo}'s regular one is that a propagator for difference constraints is registered in Line 171 and the theory description from Listing 3.5.1 is added in Line 172. Loading of the input files, grounding, and solving then follow as usual. Note that the solve function in Line 180 takes a model callback as argument. Whenever a dl-stable model is found, this callback adds symbols to it representing a mapping witnessing the satisfiability of its associated difference constraints. The additional symbols are printed as part of \texttt{clingo}'s default output.

Example 3.5.3. Running our simplified version of \texttt{clingo[dl]} using the code in Listings 3.5.4 and 3.5.5 produces the output as given in Listing 3.5.6. Observe that the two reported solutions correspond to the ones given in Figure 3.5.2 where the start times of jobs are given by the symbols over function \texttt{dl}. Furthermore, note that the mapping does not mention the term 0. The graph class makes sure that this term is always mapped to zero and omits it from the output.

3.5.4 Summary. To illustrate the theory reasoning framework of \texttt{clingo}, we have sketched the design and implementation of an ASP system extended by difference constraints. The prototypical system reflects the actual \texttt{clingo[dl]} system, which has been used to successfully solve challenging scheduling problems. One prominent example is the scheduling of trains at the Swiss railway company for a railway network of about 200km with 500 train lines [1]. The system builds upon \texttt{clingo}'s API for theory solving, which much like SAT modulo theory \cite{14}, SMT\] solving allows for extending ASP with custom theories. Theory reasoning includes the whole ASP solving process from the input language to grounding and solving. In particular, problems can be modeled using
3.7. Future Work

clingo’s high level modeling language enriched with additional domain specific constraints. While systems to extend ASPs solving capabilities exist \cite{15,105,114,117}, they rely on translations to foreign languages including the aforementioned SMT. As such, they do not benefit from the advanced solving capabilities offered by clingo, like (projected) solution enumeration\cite{91}, search for optimal models \cite{7}, or heuristic modification \cite{90}. To date, clingo\cite{dl} and clingcon \cite{12} are the only ASP-based systems with support for integer constraints that are integrated tightly into a CDCL-based search algorithm to offer high performance solving and advanced reasoning modes.

3.6 Conclusion

In this thesis, we presented major aspects of the design and implementation of the clingo ASP system. Its high-level input language allows for modeling a wide range of combinatorial search problems. The language is tailored to compactly encode problems supporting disjunctive programs with aggregates. Its semantics is firmly rooted in logic and we have shown the correctness of the grounding algorithms employed by the clingo system. Based on this foundation, we developed an application programmable interface (API) that allows for embedding clingo into applications and to implement advanced forms of reasoning. We demonstrated the usefulness of the API by developing systems that offer advanced functionality not directly available in ASP. This includes the clingo\cite{dl} system to solve problems with integer constraints and the telingo system that adds temporal operators to ASP.

3.7 Future work

In Section 3.1, we presented clingo’s input language. Its design was driven by the applications of our group and feedback from researchers. In particular, theory extensions were developed to make it possible to rapidly develop systems like clingcon, clingo\cite{dl} or telingo. This can be seen in the long development of the clingcon system. The first versions of this system were forks of clingo that were difficult to maintain because they relied on low-level interfaces and needed adjustments with each update of clingo. Recent versions build on clingo’s theory specification language (and stable API) to avoid such problems. However, currently, the specification language only allows for defining simple term grammars. Future revisions of the language might provide more expressive grammars to ease the development of theory extensions. Another route is to extend the expressiveness of the language itself. This might include adding further aggregate constructs or lifting restrictions on safety to allow for writing more compact encodings. Moreover, the clingo language contains many features not covered by the ASP-Core-2 standard. Future revisions of the standard might adopt language features from clingo.

In Section 3.3, we presented multi-shot solving allowing to incrementally ground and solve a program. A program can be extended with another program given that they are compositional. This requirement forbids recursion among both programs. It would be possible to lift this restriction. For example, incremental graph-based problems that inherently require recursion could benefit from such an extension.

In Sections 3.4 and 3.5, we presented clingo’s API for building ASP-based systems. One possible future extension that came up in applications like clingo\cite{dl} and clingcon is the need to configure the solver’s decision heuristics to select literals for branching. When
constraints are implicitly captured by theories, the solver cannot initialize its decision heuristics based on such constraints. This might be remedied by providing interfaces giving the developer of a theory the means to inform the solver about important literals. Another aspect is the grounding of theory atoms. Currently, theory atoms are grounded according to a grammar without applying on-the-fly simplifications. The size of the grounding might be reduced by providing interfaces to apply simplifications while grounding. Furthermore, variables occurring in theory atoms have to be bound by positive literals for a rule to be safe. In some theories, for example in the telingo system, terms in theory atoms refer to actual atoms that could be used for providing matches. Interfaces to more tightly integrate theory atoms into the grounding process would extend the class of safe programs and allow for writing more compact encodings.
Bibliography


