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.

Contents

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 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

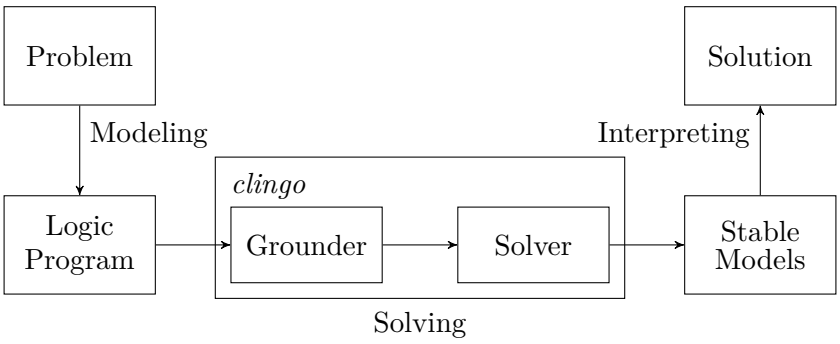

FIGURE 1.0.1. ASP solving process

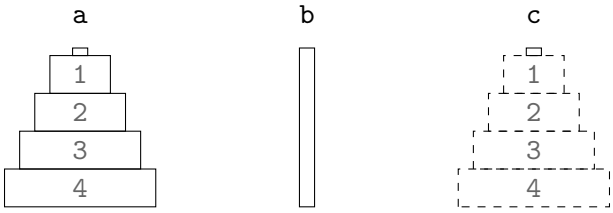

FIGURE 1.0.2. Towers of Hanoi instance with 4 discs and 3 pegs

```
#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.

`time(`t`).`	for $1 \leq t \leq 15$
`peg(`p`).`	for $p \in \{\texttt{a}, \texttt{b}, \texttt{c}\}$
`disc(`d`).`	for $1 \leq d \leq 4$
`init(`d`,a).`	for $1 \leq d \leq 4$
`goal(`d`,c).`	for $1 \leq d \leq 4$

(B) Grounded *clingo* program.

LISTING 1.0.1. Facts for ToH instance in Figure 1.0.2

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.

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 $2^{\texttt{m}} - 1 = 15$. Both constants are used throughout the program. The remaining lines in the program define facts over the predicates `time`, `peg`, `disc`, `init`, and `goal`.

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

% choose discs to move
{ move(D,P,Q,T) } :- on(D,P,T-1), peg(Q), P!=Q, time(T).

% there must be at most one move per time step.
:- time(T), #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), time(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
:- time(T), not time(T+1), goal(D,P), not on(D,P,T).

% restrict output to moves
#show move/4.
```

LISTING 1.0.2. Encoding for ToH problem

We first define facts over predicate `time` using the constant `n`. The rule uses the range term `1..n` as argument of `time`, which is expanded by the grounding component to the facts '`time(`t`).`' for $1 \leq t \leq 15$. The purpose of the `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 `a;b;c`, which is expanded to the facts '`peg(`p`).`' for $p \in \{\texttt{a}, \texttt{b}, \texttt{c}\}$. The available discs over predicate `disc` are then defined using a range term as for the time steps. Finally, the `init` and `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 `init` and `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 `on` to capture the location of a disc

at a given time point; its first argument is the disc, the second the peg, and the third the time point. Similarly, we use predicate `move` to describe moves of a disc from one peg to another at a given time point; its first argument is the disc to be moved, the second the peg it is moved from, the third the peg it is moved to, and the fourth the time point.

In Line 2, we establish the initial situation at time point `0`. The rule can be read as follows: for all possible ground terms for d and p, derive atom `on(`d`,`p`,0)` in the head of the rule whenever the atom `init(`d`,`p`)` in the body of the rule is true. When the grounder instantiates this rule for the above problem instance, it actually only produces rule instances with values for d and p where there are matching facts over `init`. We discuss this process in Section 3.2.

Next, in Line 5, we generate candidate discs to move using a choice rule as indicated by the rule head surrounded by curly braces. The solver is free to choose such rule heads whenever all elements of the rule body are true. Any disc on any peg at a previous time point is a candidate to be moved to a different peg at the current time point. Note that no move is generated at the initial time point because atoms `on(D,P,-1)` can never become true. Following the generate-define-test approach [**115**], these candidates include invalid moves; we discard such invalid moves using integrity constraints in the follow up part of the encoding. An integrity constraint discards a solution whenever all elements of its rule body are true. The integrity constraint in Line 8 ensures that there is at most one move per time step. Here we use a `#count` aggregate to count the number of candidate moves per time step. Whenever this count is larger than `1`, the solution is discarded. The following two integrity constraints discard moves where either the disc is blocked by another disc or the target peg contains a smaller disc.

At this point, we have ensured that any move leads to a valid configuration but we have not yet encoded how to transition from one time point to the next. To do this, we specify the effect of a move in Line 15. The rule states that the new location of a disc subject to a move is the target peg. The following two rules are more interesting and showcase the roles of default negation (`not`) and classical negation (`-`) in ASP. The rule in Line 17 asserts that by default the location of a disc does not change. The interesting part is the body literal `not -on(D,P,T+1)`, which is true unless it has been explicitly derived that disc `D` is not on peg `D`, that is, `-on(D,P,T+1)` is true. For this to work, the next rule derives classically false atoms over `on` asserting that a disc has a unique location, that is, a disc cannot be on a peg if it is already on some other peg.

Finally, we ensure, in Line 22, that the goal situation has been reached at the last time point and, in Line 25, that the output of the solver is restricted to atoms over predicate `move`. We depict the output when passing the above instance and encoding to *clingo* in Listing 1.0.3. By additionally passing argument `0` on the command line, we instruct the solver to enumerate all solutions to the problem. The solver reports exactly one solution consisting of 15 moves that lead from the initial to the goal situation.

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

```
$ clingo instance.lp encoding.lp 0
clingo version 5.5.2
Reading from instance.lp ...
Solving...
Answer: 1
move(1,a,b,1)  move(2,a,c,2)  move(3,a,b,4)  move(1,b,c,3)
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)
SATISFIABLE

Models  : 1
Calls   : 1
Time    : 0.02s (Solving: 0.01s 1st Model: 0.01s Unsat: 0.00s)
CPU Time: 0.02s
```

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.

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 *gringo* and the (propositional) solver *clasp*. Relying on these systems, *clingo*, *iclingo*, *claspD*, and *claspar* offer extended functionality. The *clingo* system is the monolithic combination of *clasp* and *gringo* providing the user with the convenience of not having to deal with two different programs. The *iclingo* system extends *clingo* with an incremental grounding and solving mode, which can for example be used to solve planning problems. The *claspD* solver adds support for disjunctive logic programs to *clasp* allowing the user to solve problems on the second level of the polynomial hierarchy. Finally, there is the *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 *gringo*, *clingo*, *iclingo*, and *claspar* systems as well as the design of the input languages of both *gringo* and *iclingo*. The *gringo* and *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 *claspar* system can be seen as a predecessor to the multi-threaded *clasp* system, which offers similar functionality for the nowadays ubiquitous multi-core architecture. The functionalities of *clasp*, *gringo*, and *iclingo* have meanwhile been

combined and extended resulting in the *clingo*-5 system, which we discuss in detail in this thesis. We include the paper in Section 2.1 and discuss the historical development of *clingo*'s input language in Section 3.1.

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

[1] Meanwhile 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 [**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 [**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 [**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 Σ 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 Σ, a *literal* over Σ is either the atom itself or its negation $\neg a$. A literal without negation is called *positive*, and *negative* otherwise.

A *rule* over signature Σ has form $h \leftarrow B$ where h is an atom over Σ and B is a finite set of literals over Σ. 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 Σ is a finite set of rules over Σ. 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 $L^+ = \{l \in L \mid l \text{ is an atom}\}$ 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 example 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*

- $\mathcal{P} = \{step, peg, disc, init, e, ne, o, no, m, nm, f\}$,
- $\mathcal{V} = \{P, P', D, T, T'\}$, *and*

$$\begin{aligned} step(0,1) \qquad & step(1,2) \\ peg(1) \qquad & peg(2) \\ disc(1) \qquad & disc(2) \\ init(1,1) \qquad & init(2,1) \end{aligned}$$

LISTING 3.1.1. Simplified ToH instance

$$\begin{aligned}
(r_1) \quad & e(P,P) \leftarrow peg(P) \\
(r_2) \quad & ne(P',P) \leftarrow peg(P'), peg(P), \neg e(P',P) \\
(r_3) \quad & o(D,P,0) \leftarrow init(D,P) \\
(r_4) \quad & m(D,P,T) \leftarrow \neg nm(D,P,T), o(D,P',T'), ne(P',P), step(T',T) \\
(r_5) \quad & nm(D,P,T) \leftarrow \neg m(D,P,T), o(D,P',T'), ne(P',P), step(T',T) \\
(r_6) \quad & o(D,P,T) \leftarrow m(D,P,T) \\
(r_7) \quad & o(D,P,T) \leftarrow o(D,P,T'), \neg no(D,P,T), step(T',T) \\
(r_8) \quad & no(D,P,T) \leftarrow o(D,P',T), ne(P',P) \\
(r_9) \quad & f \leftarrow \neg f, o(D,P,T), no(D,P,T)
\end{aligned}$$

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 focussing 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*

$$(g_1) \qquad e(0,0) \leftarrow peg(0)$$

$$(g_2) \qquad e(1,1) \leftarrow peg(1)$$

$$(g_3) \qquad e(2,2) \leftarrow peg(2)$$

because we have $\mathcal{C} = \{0,1,2\}$. Observe that for a rule with n variables, we obtain $|\mathcal{C}|^n$ 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 $peg(0)$, 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 Σ is the set of all ground atoms over Σ. A *two-valued interpretation* over an interpretation Σ is a set $I \subseteq \mathcal{A}$ of atoms where $\mathcal{A}$ is the atom base of Σ. Atoms contained in I are considered *true* and atoms contained in $\mathcal{A} \setminus I$ 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 $\mathcal{M} = \{1,2\} \times \{1,2\}$ *captures all possible moves of discs at time steps. Let* $M \subseteq \mathcal{M}$ *be a set of moves and* $\bar{M} = \mathcal{M} \setminus M$ *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*

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

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 `#min`, `#max`, `#count`, `#sum`, `#avg`, and `#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 '`#sum { ... } <=` *b*'. However, we retained and extended support for some language constructs introduced in *lparse*. This includes choice rules, cardinality constraints as a shortcut for `#count` aggregates, conditional literals, and term pools. Support for less frequently used aggregates like `#avg` and `#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 [**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 $\mathrm{lfp}_{\subseteq}(T_P)$. Given a program P, we define the *stable operator* S_P applied to an interpretation I as

$$S_P(I) = \mathrm{lfp}_{\subseteq}(T_{P^I}).$$

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 [**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 Σ is a pair (I, J) of sets $I \subseteq \mathcal{A}$ and $J \subseteq \mathcal{A}$ of atoms where $\mathcal{A}$ is the atom base of Σ. 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 $(I, J) \sqcup (I', J')$ as a shortcut for $(I \cup I', J \cup J')$. Furthermore, we say that (I', J') is *less precise* than (I, J), written $(I', J') \leq_{\mathrm{p}} (I, J)$, 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 Σ is $(\emptyset, \mathcal{A})$ where $\mathcal{A}$ is the atom base of Σ.

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

As put forward in [**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 [**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. $\leq_{\mathrm{p}}$, which we refer to as the *well-founded model* of P and write as

$$WM(P) = \mathrm{lfp}_{\leq_{\mathrm{p}}}(W_P).$$

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*

$$\begin{aligned} J_0 &= I, \text{ and} \\ J_i &= \{m(d,p,i), nm(d,p,i), \\ &\qquad o(d,p,i), no(d,p,i) \mid d,p \in \mathcal{C}, no(d,p,i-1) \in J_{i-1}\} \\ &\quad \cup \{o(d,p,i), no(d,p,i) \mid d,p \in \mathcal{C}, o(d,p,i-1) \in J_{i-1}\} \\ &\quad \cup \{f\}. \end{aligned}$$

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 $(I, J) \leq_{\mathrm{p}} WM(P)$. 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 $\mathrm{pred}(H(r_2)) \in \mathrm{pred}(B(r_1)^{\pm})$. Rule r_1 depends *positively* or *negatively* on r_2 if $\mathrm{pred}(H(r_2)) \in \mathrm{pred}(B(r_1)^+)$ or $\mathrm{pred}(H(r_2)) \in \mathrm{pred}(B(r_1)^-)$, 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 $(P_i)_{i \in \mathbb{I}}$ of its strongly connected components such that $i < j$ if P_j depends on P_i.

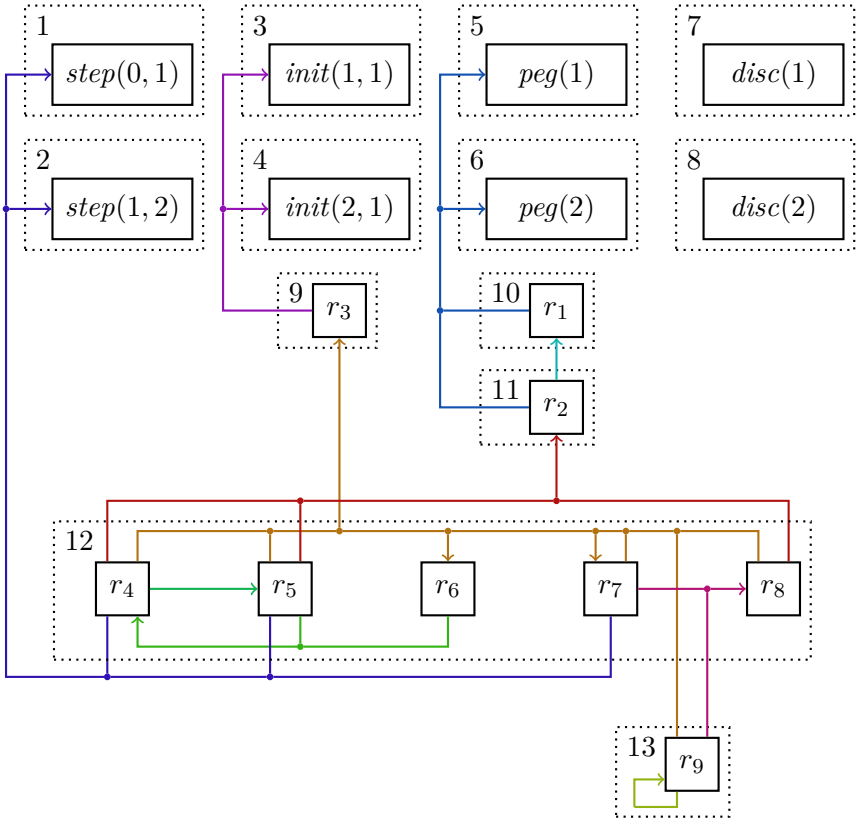

FIGURE 3.2.1. Rule dependencies of simplified ToH encoding

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

$$S_P^{IC}(J) = \mathrm{lfp}_{\subseteq}(T_{PJ}^{IC}),$$

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

$$W_P^{IC,JC}(I, J) = (S_P^{IC}(J), S_P^{JC}(I)), \text{and}$$

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

$$WM^{IC,JC}(P) = \mathrm{lfp}_{\leq_{\mathrm{p}}}(W_P^{IC,JC}).$$

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*

$$\begin{aligned} IC = JC = F & \\ \cup\, \{e(1,1), e(2,2)\} & \quad \textit{via } r_1 \\ \cup\, \{ne(1,2), ne(2,1)\} & \quad \textit{via } r_2 \\ \cup\, \{o(1,1,0), o(2,1,0)\} & \quad \textit{via } r_3. \end{aligned}$$

With this, we compute the certain atoms $I = S_P^{IC}(\mathcal{A})$ via the smallest fixed point of $T_{P^{\mathcal{A}}}^{IC}$. Observe that the reduct with the atom base $\mathcal{A}$ 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

$$I = T_{P_{12}^{\mathcal{A}}}^{IC}(\emptyset) = \{no(1,2,0), no(2,2,0)\} \quad \textit{via } r_8$$

is the least fixed point of $T_{P_{12}^{\mathcal{A}}}^{IC}$.

Next, let us compute the possible atoms $J = S_{P_{12}}^{JC}(\emptyset)$ via the smallest fixed point of $T_{P_{12}^{\emptyset}}^{JC}$. 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

$$\begin{aligned} J_0 = T_{P_{12}^{\emptyset}}^{JC}(\emptyset) &= I \\ &\cup\, \{m(1,2,1), m(2,2,1)\} && \textit{via } r_4 \\ &\cup\, \{nm(1,2,1), nm(2,2,1)\} && \textit{via } r_5 \\ &\cup\, \{o(1,1,1), o(2,1,1)\} && \textit{via } r_7, \\ J_1 = T_{P_{12}^{\emptyset}}^{JC}(J_0) &= J_0 \\ &\cup\, \{m(1,2,2), m(2,2,2)\} && \textit{via } r_4 \\ &\cup\, \{nm(1,2,2), nm(2,2,2)\} && \textit{via } r_5 \end{aligned}$$

$$\begin{aligned}
&\cup \{o(1,2,1), o(2,2,1)\} && \textit{via } r_6\\
&\cup \{o(1,1,2), o(2,1,2)\} && \textit{via } r_7\\
&\cup \{no(1,2,1), no(2,2,1)\} && \textit{via } r_8,\\
J_2 = T^{JC}_{P^{\emptyset}_{12}}(J_1) &= J_1\\
&\cup \{m(1,1,2), m(2,1,2)\} && \textit{via } r_4\\
&\cup \{nm(1,1,2), nm(2,1,2)\} && \textit{via } r_5\\
&\cup \{o(1,2,2), o(2,2,2)\} && \textit{via } r_6\\
&\cup \{no(1,1,1), no(2,1,1), no(1,2,2), no(2,2,2)\} && \textit{via } r_8,\\
J_3 = T^{JC}_{P^{\emptyset}_{12}}(J_2) &= J_2\\
&\cup \{no(1,1,2), no(2,1,2)\} && \textit{via } r_8.
\end{aligned}$$

The set $J = J_3$ is the least fixed point of $T^{JC}_{P^{\emptyset}_{12}}$.

Observe that we have just computed $(I, J) = W^{IC,JC}_{P_{12}}(\emptyset, \mathcal{A})$. This is also the least fixed point of the well-founded operator $W^{IC,JC}_{P_{12}}$. We obtain that $(I, J) = WM^{IC,JC}(P_{12})$ 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

$$(IC_i, JC_i) = \bigsqcup_{i<j}(I_i, J_i) \text{ and} \tag{1}$$

$$(I_i, J_i) = WM^{IC_i,JC_i}(P_i). \tag{2}$$

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*

components P_i:

$$I_1 = J_1 = \{step(0,1)\} \qquad I_2 = J_2 = \{step(1,2)\}$$
$$I_3 = J_3 = \{init(1,1)\} \qquad I_4 = J_4 = \{init(1,2)\}$$
$$I_5 = J_5 = \{peg(1)\} \qquad I_6 = J_6 = \{peg(2)\}$$
$$I_7 = J_7 = \{disc(1)\} \qquad I_8 = J_8 = \{disc(2)\}$$
$$I_9 = J_9 = \{o(1,1,0), o(2,1,0)\}$$
$$I_{10} = J_{10} = \{e(1,1), e(2,2)\} \qquad I_{11} = J_{11} = \{ne(1,2), ne(2,1)\}$$
$$I_{12} = \{no(1,2,0), no(2,2,0)\}$$
$$J_{12} = I_{12} \cup \{m(d,2,1), nm(d,2,1) \mid 1 \leq d \leq 2\}$$
$$\cup \{m(d,p,2), nm(d,p,2) \mid 1 \leq d,p \leq 2\}$$
$$\cup \{o(d,p,t), no(1,2,t) \mid 1 \leq d,p,t \leq 2\}$$
$$(I_{13}, J_{13}) = (\emptyset, \{f\})$$

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 $\bigsqcup_{1 \leq i \leq 11}(I_i, J_i)$ 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

$$P' = \{r \in P \mid \text{pred}(B(r)^-) \cap \text{pred}(H(P)) = \emptyset\}, \tag{3}$$

$$I = S^{IC}_{P'}(JC), \text{ and} \tag{4}$$

$$J = S^{JC}_{P}(IC \cup I). \tag{5}$$

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 $P^{\mathcal{A}}_{12}$ 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 $(\emptyset, \mathcal{A})$.*

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

$$(IC_i, JC_i) = \bigsqcup_{i<j}(I_i, J_i) \text{ and}$$
$$(I_i, J_i) = AM^{IC_i, JC_i}(P_i).$$

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 $AM((P_i)_{i \in \mathbb{I}}) \leq_{\text{p}} WM((P_i)_{i \in \mathbb{I}})$. For *stratified programs*, where no strongly connected component depends

negatively on itself, we obtain that $AM((P_i)_{i\in\mathbb{I}}) = WM((P_i)_{i\in\mathbb{I}})$. 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 $AM(P) = AM((P_i)_{i\in\mathbb{I}})$ 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 Σ is a mapping from the variables in Σ to the constants and variables in Σ. We use ι to denote the substitution mapping each variable to itself. The result of *applying* a substitution σ 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 σ and θ 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 σ is a *matcher* of expression e to variable-free expression g, if $e\sigma = g$ and σ 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 σ, a literal l, and an interpretation (I, J), we define the set of *matches* for l in (I, J) w.r.t. σ as

$$\texttt{Matches}_l^{I,J}(\sigma) = \begin{cases} \{\sigma \mid a\sigma \notin I\} & \text{if } a = \neg l, \\ \{\sigma \circ \sigma' \mid a \in J, \sigma' \in \text{match}(l\sigma, a)\} & \text{otherwise.} \end{cases}$$

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 `GroundRule` for rule instantiation is given in Algorithm 3.1. It takes a substitution σ and a set L of literals and yields a set of instances of a safe normal rule

```
1  function GroundRule^{I,J}_{r,f,J'}(σ, L):
2      if L ≠ ∅:                                              # match next
3          (G, l) ← (∅, Select_σ(L))
4          for each σ' in Matches^{I,J}_l(σ):
5              G ← G ∪ GroundRule^{I,J}_{r,f,J'}(σ', L \ {l})
6          return G
7      else if f = t or B(rσ)^+ ⊈ J':                         # rule instance
8          return {rσ}
9      else:                                                  # rule seen
10         return ∅
```

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 σ that induces an instance $r\sigma$ of rule r. To this end, $\texttt{Select}_\sigma(L)$ 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 $\texttt{Matches}^{I,J}_l(\sigma)$. Whenever an application of `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 $B(r\sigma)^+ \not\subseteq J'$ in Line 7 makes sure that the instance is discarded if it was already obtained by a previous invocation `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

- $\{r\}^{I,J} = \texttt{GroundRule}^{I,J}_{r,\mathbf{t},\emptyset}(\iota, B(r))$, and
- $\{r\}^{I,J} = \{r\}^{I,J'} \cup \texttt{GroundRule}^{I,J}_{r,\mathbf{f},J'}(\iota, B(r))$.

Thus, calls to `GroundRule` amount to applications of the one-step provability operator because $H(\{r\}^{I,J}) = T_{P^I}(J)$. 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* $\texttt{GroundRule}^{IC,JC}_{r_4,\mathbf{t},\emptyset}(\sigma, B(r_4))$ *and* $\texttt{GroundRule}^{IC,JC\cup J_0}_{r_4,\mathbf{f},J_0}(\sigma, B(r_4))$ *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.*

$o(D,P',T')$	$t(T',T)$	$ne(P',P)$	$\neg nm(D,P,T)$	$m(D,P,T)$
$o(1,1,0)$	$\longrightarrow t(0,1)$	$\longrightarrow ne(1,2)$	$\longrightarrow \neg nm(1,2,1)$	$\longrightarrow m(1,2,1)$
$\vert$				
$o(2,1,0)$	$\longrightarrow t(0,1)$	$\longrightarrow ne(1,2)$	$\longrightarrow \neg nm(2,2,1)$	$\longrightarrow m(2,2,1)$
$o(1,1,0)$	$\longrightarrow t(0,1)$	$\longrightarrow ne(1,2)$	$\longrightarrow \neg nm(1,2,1)$	$\longrightarrow \times$
$\vert$				
$o(2,1,0)$	$\longrightarrow t(0,1)$	$\longrightarrow ne(1,2)$	$\longrightarrow \neg nm(2,2,1)$	$\longrightarrow \times$
$\vert$				
$o(1,1,1)$	$\longrightarrow t(1,2)$	$\longrightarrow ne(1,2)$	$\longrightarrow \neg nm(1,2,2)$	$\longrightarrow m(1,2,2)$
$\vert$				
$o(2,1,1)$	$\longrightarrow t(1,2)$	$\longrightarrow ne(1,2)$	$\longrightarrow \neg nm(2,2,2)$	$\longrightarrow m(2,2,2)$

TABLE 1. Grounding rule r_4 obtaining atoms in J_0 and J_1 over predicate m as in Example 3.2.5

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 `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* `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 `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 `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, `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

1 **function** `GroundComponent`(P, IC, JC)**:**
2 $(G, f, J', J) \leftarrow (\emptyset, \mathbf{t}, \emptyset, \emptyset)$
3 **while** $f = \mathbf{t}$ **or** $J' \neq J$**:**
4 $G \leftarrow G \cup \bigcup_{r \in P} \texttt{GroundRule}^{IC, JC \cup J}_{r, f, JC \cup J'}(\iota, B(r))$
5 $(f, J', J) \leftarrow (\mathbf{f}, J, J \cup H(G))$
6 **return** G

Algorithm 3.2. Grounding components

1 **function** `Ground`(P)**:**
2 **let** $(P_i)_{i \in \mathbb{I}}$ be an instantiation sequence for P
3 $(F, G) \leftarrow (\emptyset, \emptyset)$
4 **for each** i **in** $\mathbb{I}$**:**
5 $P'_i \leftarrow \{r \in P_i \mid \mathrm{pred}(H(P_i)) \cap \mathrm{pred}(B(r)^-) = \emptyset\}$
6 $F \leftarrow F \cup \texttt{GroundComponent}(P'_i, H(G), H(F))$
7 $G \leftarrow G \cup \texttt{GroundComponent}(P_i, H(F), H(G))$
8 **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 $J = S_P^{JC}(IC)$ and $G = \texttt{GroundComponent}(P, IC, JC)$, we have $H(G) = J$ and $G = P^{IC, JC \cup J}$. Thus, function `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* `GroundRule` *for each rule in component* P_{12} *produce the rule instances whose heads correspond to interpretation* J *from Example 3.2.5. Function* `GroundComponent` *called with context* (IC, JC) *calls* `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* `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 `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 $\texttt{Ground}(P) = P^{AM(P)}$.

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* `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` *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* `--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 nm 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 nm, negative literals over nm always match and* `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 `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 {r1}
    e(1,1).                         e(2,2).
% component {r2}
   ne(2,1).                        ne(1,2).
% component {r3}
  o(1,1,0).                       o(2,1,0).
% component {r4, r6, r7}
  o(2,1,1) :- not no(2,1,1).     o(1,1,1) :- not no(1,1,1).
  m(2,2,1) :- not nm(2,2,1).     m(1,2,1) :- not nm(1,2,1).
  o(1,1,2) :- o(1,1,1), not no(1,1,2).
  o(2,1,2) :- o(2,1,1), not no(2,1,2).
  m(1,2,2) :- o(1,1,1), not nm(1,2,2).
  m(2,2,2) :- o(2,1,1), not nm(2,2,2).
  o(1,2,1) :- m(1,2,1).          o(2,2,1) :- m(2,2,1).
  o(2,2,2) :- o(2,2,1), not no(2,2,2).
  o(1,2,2) :- o(1,2,1), not no(1,2,2).
  m(2,1,2) :- o(2,2,1), not nm(2,1,2).
  m(1,1,2) :- o(1,2,1), not nm(1,1,2).
  o(2,2,2) :- m(2,2,2).          o(1,2,2) :- m(1,2,2).
  o(1,1,2) :- m(1,1,2).          o(2,1,2) :- m(2,1,2).
% component {r5}
 nm(1,2,1) :- not m(1,2,1).     nm(2,2,1) :- not m(2,2,1).
 nm(2,2,2) :- o(2,1,1), not m(2,2,2).
 nm(1,2,2) :- o(1,1,1), not m(1,2,2).
 nm(1,1,2) :- o(1,2,1), not m(1,1,2).
 nm(2,1,2) :- o(2,2,1), not m(2,1,2).
% component {r8}
 no(1,1,1) :- o(1,2,1).         no(2,1,1) :- o(2,2,1).
 no(2,1,2) :- o(2,2,2).         no(1,1,2) :- o(1,2,2).
 no(1,2,0).                     no(2,2,0).
 no(2,2,1) :- o(2,1,1).         no(1,2,1) :- o(1,1,1).
 no(1,2,2) :- o(1,1,2).         no(2,2,2) :- o(2,1,2).
% component {r9}
         f :- no(2,1,1), o(2,1,1), not f.
         f :- no(1,1,1), o(1,1,1), not f.
         f :- no(1,1,2), o(1,1,2), not f.
         f :- no(2,1,2), o(2,1,2), not f.
         f :- no(1,2,1), o(1,2,1), not f.
         f :- no(2,2,1), o(2,2,1), not f.
         f :- no(2,2,2), o(2,2,2), not f.
         f :- no(1,2,2), o(1,2,2), not f.
```

LISTING 3.2.1. *Clingo* output for simplified ToH example

pose one further challenge. 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 $h \in \mathcal{A}$ *depends positively* on an atom $b \in \mathcal{A}$ 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* $\mathbb{P}$ 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

$$B(P)^{\pm} \subseteq JI \cup JO \tag{6}$$

are either output or input atoms and that all head atoms

$$H(P) \subseteq JO \tag{7}$$

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 $P \cup \{a \leftarrow \mid a \in JI \cap X\}$. Two modules (P_1, JI_1, JO_1) and (P_2, JI_2, JO_2) are *compositional* if

$$JO_1 \cap JO_2 = \emptyset \text{ and} \tag{8}$$

$$JO_1 \cap C = \emptyset \text{ or } JO_2 \cap C = \emptyset \tag{9}$$

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 $\mathbb{P}_1 = (P_1, JI_1, JO_1)$

and $\mathbb{P}_2 = (P_2, JI_2, JO_2)$, we define their *join* $\mathbb{P}_1 \bowtie \mathbb{P}_2$ as

$$(P_1 \cup P_2, (JI_1 \setminus JO_2) \cup (JI_2 \setminus JO_1), JO_1 \cup JO_2). \tag{10}$$

The module theorem [**123**] shows that, for compositional modules, an interpretation X is a stable model of $\mathbb{P}_1 \bowtie \mathbb{P}_2$ 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, $\mathbb{P}_1 \bowtie \mathbb{P}_2 \bowtie \mathbb{P}_3 = (\mathbb{P}_1 \bowtie \mathbb{P}_2) \bowtie \mathbb{P}_3$ for modules $\mathbb{P}_1$, $\mathbb{P}_2$, and $\mathbb{P}_3$.

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

$$\begin{aligned} F = \{&o(1,1,0), ne(1,2), step(0,1),\\ &o(1,2,1), ne(2,1), step(1,2)\} \end{aligned}$$

of atoms to define the module

$$\mathbb{P}_0 = (\{a \leftarrow \ \mid a \in F\}, \emptyset, F).$$

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

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

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

$$\begin{aligned} \mathbb{P}_1 = (&\{g_{4,1}, g_{5,1}, g_{6,1}\},\\ &\{o(1,1,0), ne(1,2), step(0,1)\},\\ &\{m(1,2,1), nm(1,2,1), o(1,2,1)\}) \text{ and}\\ \mathbb{P}_2 = (&\{g_{4,2}, g_{5,2}, g_{6,2}\},\\ &\{o(1,2,1), ne(2,1), step(1,2)\},\\ &\{m(1,1,2), nm(1,1,2), o(1,1,2)\}). \end{aligned}$$

Observe that we can compose modules $\mathbb{P}_0 \bowtie \mathbb{P}_1 \bowtie \mathbb{P}_2$, 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 $\mathbb{P}_0$ 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 $\mathbb{P}_i$ starting from a module that represents the initial state $\mathbb{P}_0$. 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 $\mathbb{G}_1$ as

$$\Gamma\mathrm{M}(\mathbb{G}_1, P) = \mathbb{G}_2$$

where

$$\mathbb{G}_1 = (G_1, JI_1, JO_1),$$
$$\mathbb{G}_2 = (\Gamma(P), JI_2, JO_2),$$
$$JI_2 = JO_1 \setminus H(\Gamma(P)), \text{ and}$$
$$JO_2 = A(\Gamma(P)) \setminus JI_2.$$

Observe that the modules $\mathbb{G}_1$ and $\mathbb{G}_2$ are not necessarily compositional. In our current setup, where the inputs of $\mathbb{G}_2$ are solely given by the outputs of $\mathbb{G}_1$, they are compositional whenever $JO_1 \cap H(\Gamma(P)) = \emptyset$.[3] If the modules are compositional, we obtain the module $\mathbb{G}_1 \bowtie \mathbb{G}_2$, which can be extended by grounding further programs. Note that for $\mathbb{G}_1 = (\emptyset, \emptyset, \emptyset)$ the join $\mathbb{G}_1 \bowtie \mathbb{G}_2 = \mathbb{G}_2$ 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

$$\mathbb{G}_1 = (G_1, JI_1, JO_1) \text{ and}$$
$$\mathbb{G}_2 = (G_2, JI_2, JO_2) = \Gamma\mathrm{M}(\mathbb{G}_1, P)$$

be modules such that $\mathbb{G}_1$ and $\mathbb{G}_2$ 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

$$(I_1, J_1) = WM^{\emptyset, JI_1}(G_1) \text{ and}$$
$$(I_2, J_2) = WM^{(I_1,J_1)\sqcup(\emptyset,JI_1)}(G_2)$$

[3]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.

1 **function** `Ground`(P, IC, JC)**:**
2 **let** $(P_i)_{i \in \mathbb{I}}$ be an instantiation sequence for P
3 $(F, G) \leftarrow (\emptyset, \emptyset)$
4 **for each** i **in** $\mathbb{I}$**:**
5 $P'_i \leftarrow \{r \in P_i \mid \mathrm{pred}(H(P_i)) \cap \mathrm{pred}(B(r)^-) = \emptyset\}$
6 $F \leftarrow F \cup$ `GroundComponent`$(P'_i, H(G), H(F))$
7 $G \leftarrow G \cup$ `GroundComponent`$(P_i, H(F), H(G))$
8 **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

$$G'_2 = G_2^{(I_1,J_1)\sqcup(I_2,J_2)\sqcup(\emptyset,JI_1)},$$

which we associate with the module

$$\mathbb{G}'_2 = (G'_2, JI_2, A(G'_2) \setminus JI_2).$$

Using the results from Section 3.2, we obtain that the modules $\mathbb{G}_2$ and $\mathbb{G}'_2$ as well as the joins $\mathbb{G}_1 \bowtie \mathbb{G}_2$ and $\mathbb{G}_1 \bowtie \mathbb{G}'_2$ 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 $\mathbb{G}_1$ and contextual interpretation (IC, JC) as

$$\texttt{GroundModule}^{IC,JC}(\mathbb{G}_1, P) = (\mathbb{G}_2, (I_2, J_2))$$

where

$$\begin{aligned}
\mathbb{G}_1 &= (G_1, JI_1, JO_1),\\
(G_2, I_2, J_2) &= \texttt{Ground}(P, IC, JC \cup JI_1),\\
\mathbb{G}_2 &= (G_2, JI_2, JO_2),\\
JI_2 &= JO_1 \setminus H(G_2), \text{ and}\\
JO_2 &= A(G_2) \setminus JI_2.
\end{aligned}$$

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 $\mathbb{G}_1 = (G_1, JI_1, JO_1)$ 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, $\mathbb{G}_2 = \Gamma\mathrm{M}(\mathbb{G}_1, P)$, and $(\mathbb{G}'_2, (I'_2, J'_2)) = \texttt{GroundModule}^{IC,JC}(\mathbb{G}_1, P)$. If $(IC, JC) \leq_\mathrm{p} WM^{\emptyset,JI_2}(G_1)$ and $\mathbb{G}_1$ and $\mathbb{G}_2$ are compositional, then we get that

(1) $\mathbb{G}_1 \bowtie \mathbb{G}_2$ and $\mathbb{G}_1 \bowtie \mathbb{G}'_2$ have the same stable models, and
(2) $(IC, JC) \sqcup (I'_2, J'_2) \leq_\mathrm{p} WM^{\emptyset,JI'}(G')$ where $(G', JI', JO') = \mathbb{G}_1 \bowtie \mathbb{G}'_2$.

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 $\mathbb{G}_1 \bowtie \mathbb{G}_2$ are a superset of the outputs of $\mathbb{G}_1 \bowtie \mathbb{G}'_2$ in general. Property (2) shows that we can use $(IC, JC) \sqcup (I'_2, J'_2)$ as contextual interpretation to further extend the module $\mathbb{G}_1 \bowtie \mathbb{G}'_2$.

To be able to ground compositional modules from a fixed set of programs, we define *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 *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 $P[\boldsymbol{k}/\boldsymbol{c}]$ instead of $P[k_1/c_1, \ldots, k_n/c_n]$ for tuples $\boldsymbol{k} = (k_1, \ldots, k_n)$ and $\boldsymbol{c} = (c_1, \ldots, c_n)$ 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

$$P_\mathrm{base} = F \cup \{r_1, r_2, r_3\}$$

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

$$P_\mathrm{step} = \{r'_4, r'_5, r'_6\}$$

be a program with parameters t and t', and

$$P_\mathrm{check} = \{r'_8, r'_9\}$$

be a program with parameter t.

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

$$(\mathbb{G}_0, (I_0, J_0)) = \texttt{GroundModule}^{\emptyset,\emptyset}((\emptyset,\emptyset,\emptyset), P_{\text{base}} \cup P_{\text{check}}[t/0]) \text{ and}$$
$$\mathbb{P}_0 = (\emptyset,\emptyset,\emptyset) \bowtie \mathbb{G}_0 = \mathbb{G}_0$$

by extending the empty module. We obtain that module

$$\mathbb{G}_0 = (G_0, \emptyset, JO_0)$$

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

$$(IC_i, JC_i) = \bigsqcup_{j<i} (I_j, J_j),$$
$$(\mathbb{G}_i, (I_i, J_i)) = \texttt{GroundModule}^{IC_i, JC_i}(\mathbb{P}_{i-1}, P_{\text{step}}[t'/(i-1), t/i] \cup P_{\text{check}}[t/i]), \text{ and}$$
$$\mathbb{P}_i = \mathbb{P}_{i-1} \bowtie \mathbb{G}_i.$$

First, we verify that $\mathbb{P}_{i-1}$ *and* $\mathbb{G}_i = (G_i, JI_i, JO_i)$ *are compositional. This is ensured in the way parameters* t *and* t' *are used in the programs* P_{step} *and* P_{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* $\mathbb{P}_{i-1}$ *and the outputs of module* $\mathbb{G}_i$ *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* $JI_i = JO_0 \cup \cdots \cup JO_{i-1}$*. We obtain that*

$$\mathbb{P}_i = (G_0 \cup \cdots \cup G_i, \emptyset, JO_0 \cup \cdots \cup JO_i).$$

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 3.1.3. 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 $\Sigma = (\mathcal{C}, \mathcal{P} \cup \{\epsilon\}, \mathcal{V})$ with a fresh nullary predicate ϵ. An *extensible* program over signature Σ can use atom ϵ 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 $\mathbb{G}_1$ and contextual interpretation (IC, JC) as

$$\Gamma\text{M}(\mathbb{G}_1, P) = \mathbb{G}_2$$

where

$$\begin{aligned}
\mathbb{G}_1 &= (G_1, JI_1, JO_1),\\
\mathbb{G}_2 &= (G_2, JI_2, JO_2),\\
G_2 &= \{g \in \Gamma(P) \mid \epsilon \notin B(g)\},\\
JI_2 &= (JO_1 \cup H(\Gamma(P))) \setminus H(G_2), \text{ and}\\
JO_2 &= A(G_2) \setminus JI_2.
\end{aligned}$$

Note that all rules referring to ϵ 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 $\mathbb{G}_1$ and contextual interpretation (IC, JC) as

$$\texttt{GroundModule}^{IC,JC}(\mathbb{G}_1, P) = (\mathbb{G}_2, (I_2, J_2))$$

where

$$\begin{aligned}
\mathbb{G}_1 &= (G_1, JI_1, JO_1),\\
\mathbb{G}_2 &= (G_2, JI_2, JO_2),\\
(G_2', (I_2, J_2)) &= \texttt{Ground}(P, IC, JC \cup JI_1 \cup \{\epsilon\})\\
G_2 &= \{g \in G_2' \mid \epsilon \notin B(g)\},\\
JI_2 &= (JO_1 \cup H(G_2')) \setminus H(G_2), \text{ and}\\
JO_2 &= A(G_2) \setminus JI_2.
\end{aligned}$$

Next, we shed some light on joins with ground modules and contextual ground modules for extensible programs. Let $\mathbb{G}_1 = (G_1, JI_1, JO_1)$ be a module, (IC, JC) be an interpretation, P be an extensible program, $\mathbb{G}_2 = \Gamma\mathrm{M}(\mathbb{G}_1, P)$, and $(\mathbb{G}_2', (I_2', J_2')) = \texttt{GroundModule}^{IC,JC}(\mathbb{G}_1, P)$. If $(IC, JC) \leq_\mathrm{p} WM^{\emptyset, JI_2}(G_1)$, and $\mathbb{G}_1$ and $\mathbb{G}_2$ are compositional, then we get

(1) for interpretations X with $X \cap (JI_2 \setminus JI_2') = \emptyset$, that X is a stable model of $\mathbb{G}_1 \bowtie \mathbb{G}_2$ iff X is a stable model of $\mathbb{G}_1 \bowtie \mathbb{G}_2'$, where JI and JI' are the inputs of $\mathbb{G}_1 \bowtie \mathbb{G}_2$ and $\mathbb{G}_1 \bowtie \mathbb{G}_2'$, respectively, and
(2) $(IC, JC) \sqcup (I_2', J_2') \leq_\mathrm{p} WM^{\emptyset, JI'}(G')$ where $(G', JI', JO') = \mathbb{G}_1 \bowtie \mathbb{G}_2'$.

For extensible programs, we no longer get that $\mathbb{G}_1 \bowtie \mathbb{G}_2'$ and $\mathbb{G}_1 \bowtie \mathbb{G}_2$ have the same stable models. This is because the inputs of $\mathbb{G}_2'$ are subject to grounding. Thus, the join $\mathbb{G}_1 \bowtie \mathbb{G}_2'$ might have fewer input atoms than $\mathbb{G}_1 \bowtie \mathbb{G}_2$. 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 $(IC, JC) \sqcup (I_2', J_2')$ as contextual interpretation to further extend the module $\mathbb{G}_1 \bowtie \mathbb{G}_2'$.

EXAMPLE 3.3.3. *We extend program P_{check} from Example 3.3.2 with the following rules:*

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

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

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

Module $\mathbb{P}'_0$ has no stable models where atom $query(0)$ is true. Module $\mathbb{P}'_1$ has exactly one stable model where atom $query(1)$ is true and atom $query(0)$ 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 $\mathbb{P}'_i = (G, JI, JO)$ 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 $G \cup \{ query(i) \leftarrow \}$.

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 $(\boldsymbol{P}, \mathbb{P}, (IC, JC), A)$ where

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

When solving with $\boldsymbol{P}$, 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*) : $\mapsto (\boldsymbol{P}, \mathbb{P}, (IC, JC), A)$
for a collection $\boldsymbol{P}$ of parametrizable extensible programs read from the given list of files where

- $\mathbb{P} = (\emptyset, \emptyset, \emptyset)$,
- $(IC, JC) = (\emptyset, \emptyset)$, 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.

```
1  #program base.
2  % establish initial situation
3  on(D,P,0) :- init(D,P).
4
5  #program check(t).
6  % uniqueness of location: a disc can only be on one peg
7  -on(D,Q,t) :- on(D,P,t), peg(Q), P!=Q.
8
9  % ensure that the goal was reached
10 :- query(t), goal(D,P), not on(D,P,t).
11
12 #program step(t).
13 % choose discs to move
14 { move(D,P,Q,t) } :- on(D,P,t-1), peg(Q), P!=Q.
15
16 % there must be at most one move per time step.
17 :- #count { D,P,Q: move(D,P,Q,t) } > 1.
18 % only the topmost disc can be moved
19 :- move(D,P,_,t), on(E,P,t-1), D>E.
20 % a disc can only be put on larger discs
21 :- move(D,_,Q,t), on(E,Q,t-1), D>E.
22
23 % effects: change the location of the moved disc
24 on(D,Q,t) :- move(D,_,Q,t).
25 % inertia: discs stay in place by default
26 on(D,P,t) :- on(D,P,t-1), not -on(D,P,t).
27
28 % restrict output to moves
29 #show move/4.
```

LISTING 3.3.2. Incremental ToH encoding

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_{base} *without parameters, and programs* P_{check} *and* P_{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*

```
1  #script (python)
2
3  from clingo.symbol import Number, Function
4
5  def main(ctl):
6      step, ret, query = 0, None, None
7      while ret is None or not ret.satisfiable:
8          parts = [("check", [Number(step)])]
9          if query is not None:
10             ctl.release_external(query)
11             parts.append(("step", [Number(step)]))
12         else:
13             parts.append(("base", []))
14         ctl.ground(parts)
15         query = Function("query", [Number(step)])
16         ctl.assign_external(query, True)
17         ret, step = ctl.solve(), step + 1
18 #end.
19
20 #program check(t).
21 #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} *from Example 3.3.3 and is part of the* P_{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* $(P_n)_{n\in N}$ *for* $N = \{\text{base}, \text{check}, \text{step}\}$. *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_{base}. *Thus, the initial program state is*

$$((P_n)_{n\in N}, (\emptyset, \emptyset, \emptyset), (\emptyset, \emptyset), \emptyset).$$

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 $\boldsymbol{P}$ 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.[4]

ground$((n, \boldsymbol{c}_n)_{n\in N'}) : (\boldsymbol{P}_1, \mathbb{P}_1, (IC_1, JC_1), A_1) \mapsto (\boldsymbol{P}_2, \mathbb{P}_2, (IC_2, JC_2), A_2)$
for a collection of program names and tuples of constants to ground where

- $\boldsymbol{P}_2$ is the collection of programs obtained from $\boldsymbol{P}_1$ by extending the signatures of its programs with the constants in $\boldsymbol{c}_n$ for all $n \in N'$,
- $(\mathbb{G}, I, J) = \texttt{GroundModule}^{IC_1,JC_1}(\mathbb{P}_1, \bigcup_{n\in N'} P_n[\boldsymbol{k}_n/\boldsymbol{c}_n])$ where $\boldsymbol{k}_n$ are the parameters of program P_n and $((P_n)_{n\in N}) = \boldsymbol{P}_2$,
- $\mathbb{P}_2 = \mathbb{P}_1 \bowtie \mathbb{G}$,
- $(IC_2, JC_2) = (IC_1, JC_1) \sqcup (I, J)$, and
- $A_2 = A_1 \cap JI$ where JI are the inputs of $\mathbb{P}_2$.

Note that the join $\mathbb{P}_1 \bowtie \mathbb{G}$ turns some of the input atoms of $\mathbb{P}_1$ 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 $\mathbb{P}_2$.

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

assign$(a, v) : (\boldsymbol{P}, \mathbb{P}, (IC, JC), A_1) \mapsto (\boldsymbol{P}, \mathbb{P}, (IC, JC), A_2)$
for a ground atom a and a truth value $v \in \{\mathbf{t}, \mathbf{f}\}$ where

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

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.

release$(a) : (\boldsymbol{P}, \mathbb{P}_1, (IC, JC), A_1) \mapsto (\boldsymbol{P}, \mathbb{P}_2, (IC, JC), A_2)$
for a ground atom a where

- $A_2 = A_1 \setminus \{a\}$ and
- $\mathbb{P}_2 = \begin{cases} \mathbb{P}_1 \bowtie (\emptyset, \emptyset, \{a\}) & \text{if a is an input of } \mathbb{P}_1, \\ \mathbb{P}_1 & \text{otherwise.} \end{cases}$

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.

solve$() : (\boldsymbol{P}, \mathbb{P}, (IC, JC), A) \mapsto (\boldsymbol{P}, \mathbb{P}, (IC, JC), A)$
outputs all stable models X of $\mathbb{P}$ such that $A \subseteq X$ and $X \cap (JI \setminus A) = \emptyset$ where JI are the inputs of $\mathbb{P}$.

[4]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.

```
$ 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 $\mathbb{P} \bowtie (\{a \leftarrow \mid a \in A\}, \emptyset, JI)$ where JI 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_{base} *and* P_{check} *for time step* 0 *and then grounding programs* P_{step} *and* P_{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, `ground` *is called with the list* `[("base", []), ("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 $query(0)$ *to enable/disable the goal check is false. Before solving, we change its truth value in Line 16 to true and enter the state*

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

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 $query(i-1)$ entering state

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

and then call `ground` *with the list* `[("step", [Number(i)]), ("check", Number(i))]`*. The* `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

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

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 `Solving...` *for each* `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 move) corresponds exactly to the one in Listing 1.0.3.*

3.3.3 Summary. In Section 2.3, we present *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 *clingo* API. The procedural control is exercised via traditional programming languages. Currently, the *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 *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**, LTL$_f$]. 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 $\odot \in \{\vee, \wedge, \rightarrow, \boxdot, \diamond\!\!\!\!\circ, \blacksquare, \blacklozenge\}$.

We refer to $\bullet$, $\blacksquare$ and $\blacklozenge$ as *previous*, *trigger* and *since*, and to $\circ$, $\boxdot$ and $\diamond\!\!\!\!\circ$ 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:

$\blacksquare F = \bot \blacksquare F$	*always before*	$\boxdot F = \bot \boxdot F$	*always afterward*
$\blacklozenge F = \top \blacklozenge F$	*eventually before*	$\diamond F = \top \diamond\!\!\!\!\circ F$	*eventually afterward*
$\mathsf{I} = \neg\bullet\top$	*initial*	$\mathbb{F} = \neg\circ\top$	*final*
$\hat{\blacklozenge} F = \blacklozenge(\mathsf{I} \wedge F)$	*initially*	$\hat{\diamond} F = \diamond(\mathbb{F} \wedge F)$	*finally*
$\hat{\bullet} F = \mathsf{I} \vee \bullet F$	*weak previous*	$\hat{\circ} F = \mathbb{F} \vee \circ F$	*weak next*

A (finite) *trace* of length $n \geq 0$ over an alphabet $\mathcal{A}$ is a sequence $(X)_{0 \leq i \leq n}$ 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.

Let Σ be a signature. A *temporal atom* over Σ has form a, $\bullet a$, or $\hat{\blacklozenge} a$ where a is an atom over Σ. A *temporal literal* over Σ is either formula $\mathbb{F}$, or has form a or $\neg a$ for a temporal atom a over Σ. A *temporal program* over Σ consists of *temporal rules* over Σ of form

$$h \leftarrow B \tag{11}$$

$$\hat{\circ}\Box(h \leftarrow B) \tag{12}$$

$$\Box(h \leftarrow B) \tag{13}$$

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

We define instances of temporal rules in 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 ϕ as follows:

$$\begin{aligned}\phi(h \leftarrow B) &= \bigwedge_{l \in B} l \rightarrow h,\\ \phi(\Box r) &= \Box\phi(r),\\ \phi(\hat{\circ}\Box r) &= \hat{\circ}\phi(\Box r), \text{ and}\\ \phi(P) &= \{\phi(r) \mid r \in \Gamma(P)\}.\end{aligned}$$

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:

$$\hat{\circ}\Box(o(D,P) \leftarrow \bullet o(D,P), \neg no(D,P)).$$

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:

$$\Box(f \leftarrow \neg f, o(D,P), no(D,P)).$$

Note that an always rule $\Box r$ can equivalently be written using the initial and dynamic rules r and $\hat{\circ}\Box r$.

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 Σ into a collection of parametrizable programs over signature Σ'. The signature Σ' is obtained from Σ by incrementing the arities of predicates in Σ by one. Furthermore, we add a fresh

predicate f with arity 1 and fresh constants t and t' for parameters. We define the translation τ as follows:

$$\begin{aligned}
\tau(p(\boldsymbol{t})) &= p(\boldsymbol{t}, t),\\
\tau(\bullet p(\boldsymbol{t})) &= p(\boldsymbol{t}, t'),\\
\tau(\hat{\blacklozenge} p(\boldsymbol{t})) &= p(\boldsymbol{t}, 0),\\
\tau(\mathbb{F}) &= f(t),\\
\tau(\neg a) &= \neg\tau(a),\\
\tau(h \leftarrow B) &= \tau(h) \leftarrow \{\tau(l) \mid l \in B\}, \text{ and}\\
\tau(P) &= (P_n)_{n\in\{\text{init,dynamic,always}\}}
\end{aligned}$$

where

$$\begin{aligned}
P_{\text{init}} &= \{\tau(r) \mid r \in P\},\\
P_{\text{dynamic}} &= \{\tau(r) \mid \hat{\circ}\Box r \in P\}, \text{ and}\\
P_{\text{always}} &= \{\tau(r) \mid \Box r \in P\} \cup \{f(t) \leftarrow \epsilon\}.
\end{aligned}$$

are parametrizable extensible programs with parameters t and t'.

Next, we use the translation τ 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

$$\begin{aligned}
S_0'' &= (\phi(P), (\emptyset, \emptyset, \emptyset), (\emptyset, \emptyset), \emptyset)\\
S_0' &= \text{ground}(((\text{init}, (-1, 0)), (\text{always}, (-1, 0))))(S_0''),\\
S_0 &= \text{assign}(f(0), \mathbf{t})(S_0')
\end{aligned}$$

and for $i > 0$

$$\begin{aligned}
S_i'' &= \text{release}(f(i-1))(S_{i-1}),\\
S_i' &= \text{ground}(((\text{always}, (i-1, i)), (\text{dynamic}, (i-1, i))))(S_i''),\\
S_i &= \text{assign}(f(i), \mathbf{t})(S_i'').
\end{aligned}$$

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 solve(S_n). That is, a temporal stable model $(X_i)_{0\leq i\leq n}$ corresponds to the stable model $\{p(\boldsymbol{t}, i) \mid p(\boldsymbol{t}) \in X_i, 0 \leq i \leq n\} \cup \{f(n)\}$ as output by 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_{step} in Example 3.3.2. As compared to the

$$(r_3'') \quad o(D,P) \leftarrow \mathit{init}(D,P)$$
$$(r_4'') \quad \hat{\circ}\Box(m(D,P) \leftarrow \neg nm(D,P), \bullet o(D,P'), \hat{\blacklozenge} ne(P',P))$$
$$(r_5'') \quad \hat{\circ}\Box(nm(D,P) \leftarrow \neg m(D,P), \bullet o(D,P'), \hat{\blacklozenge} ne(P',P))$$
$$(r_6'') \quad \hat{\circ}\Box(o(D,P) \leftarrow m(D,P))$$
$$(r_7'') \quad \hat{\circ}\Box(o(D,P) \leftarrow \bullet o(D,P), \neg no(D,P))$$
$$(r_8'') \quad \Box(no(D,P) \leftarrow o(D,P'), \hat{\blacklozenge} ne(P',P))$$
$$(r_9'') \quad \Box(f \leftarrow \neg f, o(D,P), no(D,P))$$

LISTING 3.4.1. Simplified temporal ToH encoding

standard rules, we omit time parameters in these rules applying operator $\bullet$ to literals referring to the previous time point and operator $\hat{\blacklozenge}$ 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 P_{always}.

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.

3.4.2 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 $\bullet$ and $\hat{\blacklozenge}$ operators are represented by preceding an atom with a prime and an underscore, respectively. To encode the $\mathbb{F}$ 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 $\bullet$ and $\hat{\blacklozenge}$, we change the semantics of some of clingo's identifiers. For example, in Line 15, we write `'on(D,P)` *instead of $\bullet on(D,P)$ and, in Line 7, we write* `_on(D,P)` *instead of $\hat{\blacklozenge} on(D,P)$. In Line 11, we write* `&final` *instead of $\mathbb{F}$ 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

```
#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).

% restrict output to moves
#show move/3.
```

LISTING 3.4.2. Temporal ToH encoding

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 τ 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

```
1  from clingo.symbol import Function, Number
2  import clingo.ast as ast
3
4  class AtomTransformer(ast.Transformer):
5      def _prm(self, fun, loc):
6          if fun.startswith('_'):
7              return fun[1:], ast.SymbolicTerm(loc, Number(0))
8
9          if fun.startswith("'"):
10             prm = ast.BinaryOperation(
11                 loc, ast.BinaryOperator.Minus,
12                 ast.SymbolicTerm(loc, Function('__t')),
13                 ast.SymbolicTerm(loc, Number(1)))
14             return fun[1:], prm
15
16         return fun, ast.SymbolicTerm(loc, Function('__t'))
17
18     def visit_Function(self, trm):
19         fun, prm = self._prm(trm.name, trm.location)
20         trm = trm.update(name=fun)
21         trm.arguments.append(prm)
22         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 τ 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 $\hat{\blacklozenge}$ 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 `ProgramTransfomer` 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

```
25 import clingo.ast as ast
26
27 class ProgramTransformer(ast.Transformer):
28     def __init__(self):
29         self.atf = AtomTransformer()
30
31     def visit_SymbolicAtom(self, atm):
32         return atm.update(symbol=self.atf(atm.symbol))
33
34     def visit_TheoryAtom(self, atm):
35         if atm.term.name == 'final':
36             loc = atm.location
37             prm = ast.SymbolicTerm(loc, Function('__t'))
38             fun = ast.Function(loc, "__final", [prm], False)
39             return ast.SymbolicAtom(fun);
40         return atm
41
42     def visit_Program(self, prg):
43         nms = ('base', 'initial', 'always', 'dynamic')
44         if prg.name in nms:
45             pms = [ast.Id(prg.location, '__t')]
46             return prg.update(parameters=pms)
47         return prg
48
49     def visit_ShowSignature(self, sig):
50         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 $\bullet on(D, P)$ is translated to `on(D,P,__t)`. Function `visit_TheoryAtom` in Lines 34 to 40 replaces the theory atom `&final` to represent $\mathbb{F}$ 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)]))
                pts.append(('initial', [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

```
$ python telingo.py instance.lp encoding.lp 0
telingo version 2.1.2
Reading from instance.lp ...
Solving...    Solving...    Solving...    Solving...
Solving...    Solving...    Solving...    Solving...
Solving...    Solving...    Solving...    Solving...
Solving...    Solving...    Solving...    Solving...
Answer: 1
 State 0:                 State 1:  move(1,a,b)
 State 2:  move(2,a,c)    State 3:  move(1,b,c)
 State 4:  move(3,a,b)    State 5:  move(1,c,a)
 State 6:  move(2,c,b)    State 7:  move(1,a,b)
 State 8:  move(4,a,c)    State 9:  move(1,b,c)
 State 10: move(2,b,a)    State 11: move(1,c,a)
 State 12: move(3,b,c)    State 13: move(1,a,b)
 State 14: move(2,a,c)    State 15: move(1,b,c)
SATISFIABLE

Models  : 1
Calls   : 16
Time    : 0.10s (Solving: 0.04s 1st Model: 0.01s Unsat: 0.04s)
CPU Time: 0.10s
```

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 $\mathbb{F}$. 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 3.3.4 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`.

3.4.3 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

the following, we develop a simplified version of the *clingo*[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

$$\boldsymbol{a} - \boldsymbol{b} \leq c \tag{14}$$

where $\boldsymbol{a}$ and $\boldsymbol{b}$ 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 λ from tuples of constants to integers such that for all dl-atoms of form (14) in the interpretation, we have $\lambda(\boldsymbol{a}) - \lambda(\boldsymbol{b}) \leq c$. 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 λ 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 $d_{3,2} = 5$.

	durations	
tasks	*unit* u_1	*unit* u_2
t_1		
t_2		
t_3		

FIGURE 3.5.1. Flowshop instance with bound 16

unit	*first solution*	*second solution*
	$t_2 < t_3 < t_1$	$t_2 < t_1 < t_3$
u_1		
u_2		

FIGURE 3.5.2. Solutions for flowshop example

We represent this problem using the following facts:

$b(16)$	$t(1)$	$t(2)$	$t(3)$
$u(1)$	$d(1,1,3)$	$d(2,1,1)$	$d(3,1,5)$
$u(2)$	$d(1,2,4)$	$d(2,2,6)$	$d(3,2,5)$
$ord(1,2)$	$ord(2,3)$		

We use atoms $t(i)$, $u(j)$, $d(i,j,d_{ij})$, *and* $b(b)$ *to capture tasks* t_i, *units* u_j, *durations* $d_{i,j}$, *and budget* b, *respectively. Furthermore, we use* $ord(i,i+1)$ *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:

$$(r_1) \quad a(S,T) \leftarrow \neg na(S,T), t(T), t(S)$$
$$(r_2) \quad na(S,T) \leftarrow \neg a(S,T), t(T), t(S)$$
$$(r_3) \quad assigned(S) \leftarrow a(S,T)$$
$$(r_4) \quad f \leftarrow \neg f, t(S), \neg assigned(S)$$
$$(r_5) \quad eq(T,T) \leftarrow t(T)$$
$$(r_6) \quad f \leftarrow \neg f, a(S,T), a(S,T'), \neg eq(T,T')$$
$$(r_7) \quad (S,U) - (S',U) \leq -D \leftarrow a(S,T), t(S'), ord(S,S'), d(T,U,D)$$
$$(r_8) \quad (S,U) - (S,U') \leq -D \leftarrow a(S,T), u(U'), ord(U,U'), d(T,U,D)$$
$$(r_9) \quad (s) - (S,U) \leq 0 \leftarrow t(S), u(U)$$
$$(r_{10}) \quad (S,U) - (e) \leq -D \leftarrow t(S), u(U)$$
$$(r_{11}) \quad (e) - (s) \leq B \leftarrow b(B)$$

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

```
1  #theory dl {
2      term {
3          -  : 3, unary;
4          ** : 2, binary, right;
5          *  : 1, binary, left;
6          /  : 1, binary, left;
7          \\ : 1, binary, left;
8          +  : 0, binary, left;
9          -  : 0, binary, left
10     };
11     &diff/0 : term, {<=}, term, head
12 }.
```

LISTING 3.5.1. Difference logic theory

```
1  b(16).     u(1).       u(2).
2  t(1).    d(1,1,3).   d(1,2,4).
3  t(2).    d(2,1,1).   d(2,2,6).
4  t(3).    d(3,1,5).   d(3,2,5).
```

LISTING 3.5.2. Facts for flowshop instance in Figure 3.5.1

associative. Using this theory term definition, we can construct *theory terms* like

`(S,U)-(s,U+1)` 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 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 `diff` with arity 0 as name using theory terms specified by theory term definition `term` as elements. Furthermore, we can use `<=` and a theory term specified by theory term definition `term` as guard. Finally, the theory atom is restricted to occur in rule heads via label `head`. Using this theory atom definition, we can construct *theory atoms* like

`&diff { (S,U)-(s,U+1) } <= -D` or
`&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.*

```
1  1 { a(S,T) : t(S) } 1 :- t(T).
2  1 { a(S,T) : t(S) } 1 :- t(S).
3
4  &diff { (S,U)-(S+1,U) } <= -D  :- a(S,T), d(T,U,D), t(S+1).
5  &diff { (S,U)-(S,U+1) } <= -D  :- a(S,T), d(T,U,D), u(U+1).
6  &diff {     0-(S,U)   } <= 0   :- a(S,T), u(U).
7  &diff { (S,U)-0       } <= B-D :- a(S,T), d(T,U,D), b(B).
8
9  #show a/2.
```

LISTING 3.5.3. Encoding for flowshop problem

In Lines 1 to 2, we encode the assignment from tasks to slots using aggregates. The rules in line Lines 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 `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 `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 `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 `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 `_l2e` 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 `init` in Lines 119 to 128 sets up watches as well as the dictionaries `_l2e` 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

```
160 class DLApp(Application):
161     program_name = "clingo-dl"
162     version = "1.0"
163
164     def __init__(self):
165         self._propagator = DLPropagator()
166
167     def on_model(self, model):
168         self._propagator.on_model(model)
169
170     def main(self, ctl, files):
171         ctl.register_propagator(self._propagator)
172         ctl.load("dl-theory.lp")
173
174         for path in files:
175             ctl.load(path)
176         if not files:
177             ctl.load("-")
178
179         ctl.ground([("base", [])])
180         ctl.solve(on_model=self.on_model)
181
182
183 sys.exit(int(clingo_main(DLApp(), sys.argv[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.

```
$ python script.py instance.lp encoding.lp 0
clingo-dl version 1.0
Reading from instance.lp ...
Solving...
Answer: 1
dl((1,1),0) dl((2,1),1) dl((3,1),6)
dl((1,2),1) dl((2,2),7) dl((3,2),12)
a(1,2) a(2,3) a(3,1)
Answer: 2
dl((1,1),0) dl((2,1),1) dl((3,1),4)
dl((1,2),1) dl((2,2),7) dl((3,2),11)
a(1,2) a(2,1) a(3,3)
SATISFIABLE

Models  : 2
Calls   : 1
Time    : 0.02s (Solving: 0.01s 1st Model: 0.0s Unsat: 0.0s)
CPU Time: 0.02s
```

LISTING 3.5.6. *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 *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 *clingo*'s default output.

EXAMPLE 3.5.3. *Running our simplified version of 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* `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 *clingo*, we have sketched the design and implementation of an ASP system extended by difference constraints. The prototypical system reflects the actual *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 *clingo*'s API for theory solving, which much like SAT modulo theory [**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

clingo's high level modeling language enriched with additional domain specific constraints. While systems to extend ASPs solving capabilities exist [**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[**91**], search for optimal models [**7**], or heuristic modification [**90**]. To date, *clingo*[DL] and *clingcon* [**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*[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*[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*[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

[1] D. Abels, J. Jordi, M. Ostrowski, T. Schaub, A. Toletti, and P. Wanko, *Train scheduling with hybrid ASP*, Theory and Practice of Logic Programming **21** (2021), no. 3, 317–347.

[2] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of databases*, Addison-Wesley, 1995.

[3] F. Aguado, P. Cabalar, M. Diéguez, G. Pérez, and C. Vidal, *Temporal equilibrium logic: a survey*, Journal of Applied Non-Classical Logics **23** (2013), no. 1-2, 2–24.

[4] I. Aini, A. Saptawijaya, and S. Aminah, *Bringing answer set programming to the next level: A real case on modeling course timetabling*, International conference on advanced computer science and information systems, May 2018, pp. 471–476.

[5] M. Alviano, F. Calimeri, G. Charwat, M. Dao-Tran, C. Dodaro, G. Ianni, T. Krennwallner, M. Kronegger, J. Oetsch, A. Pfandler, J. Pührer, C. Redl, F. Ricca, P. Schneider, M. Schwengerer, L. Spendier, J. Wallner, and G. Xiao, *The fourth answer set programming competition: Preliminary report*, Proceedings of the twelfth international conference on logic programming and nonmonotonic reasoning (lpnmr'13), 2013, pp. 42–53.

[6] M. Alviano, W. Faber, and M. Gebser, *Rewriting recursive aggregates in answer set programming: Back to monotonicity*, Theory and Practice of Logic Programming **15** (2015), no. 4-5, 559–573.

[7] B. Andres, B. Kaufmann, O. Matheis, and T. Schaub, *Unsatisfiability-based optimization in clasp*, Technical communications of the twenty-eighth international conference on logic programming (iclp'12), 2012, pp. 212–221.

[8] B. Andres, D. Rajaratnam, O. Sabuncu, and T. Schaub, *Integrating ASP into ROS for reasoning in robots*, Proceedings of the thirteenth international conference on logic programming and nonmonotonic reasoning (lpnmr'15), 2015, pp. 69–82.

[9] J. Babb and J. Lee, *Cplus2ASP: Computing action language C+ in answer set programming*, Proceedings of the twelfth international conference on logic programming and nonmonotonic reasoning (lpnmr'13), 2013, pp. 122–134.

[10] M. Balduccini, Y. Lierler, and S. Woltran (eds.), *Proceedings of the fifteenth international conference on logic programming and nonmonotonic reasoning (lpnmr'19)*, Lecture Notes in Artificial Intelligence, vol. 11481, Springer-Verlag, 2019.

[11] M. Banbara, K. Inoue, B. Kaufmann, T. Okimoto, T. Schaub, T. Soh, N. Tamura, and P. Wanko, *teaspoon: Solving the curriculum-based course timetabling problems with answer set programming*, Annals of Operations Research **275** (2019), no. 1, 3–37.

[12] M. Banbara, B. Kaufmann, M. Ostrowski, and T. Schaub, *Clingcon: The next generation*, Theory and Practice of Logic Programming **17** (2017), no. 4, 408–461.

[13] C. Baral, *Knowledge representation, reasoning and declarative problem solving*, Cambridge University Press, 2003.

[14] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli, *Satisfiability modulo theories*, Handbook of satisfiability, 2009, pp. 825–885.

[15] M. Bartholomew and J. Lee, *System aspmt2smt: Computing ASPMT theories by SMT solvers*, Proceedings of the fourteenth european conference on logics in artificial intelligence (jelia'14), 2014, pp. 529–542.

[16] R. Bertolucci, A. Capitanelli, C. Dodaro, N. Leone, M. Maratea, F. Mastrogiovanni, and M. Vallati, *Manipulation of articulated objects using dual-arm robots via answer set programming*, Theory and Practice of Logic Programming **21** (2021), no. 3, 372–401.

[17] V. Besin, M. Hecher, and S. Woltran, *Body-decoupled grounding via solving: A novel approach on the asp bottleneck*, Proceedings of the thirty-first international joint conference on artificial intelligence, IJCAI-22, 20227, pp. 2546–2552.

[18] N. Bidoit and C. Froidevaux, *Minimalism subsumes default logic and circumscription in stratified logic programming*, Proceedings of the second annual symposium on logic in computer science (lics'87), 1987, pp. 89–97.

[19] R. Bisiani, M. Gebser, H. Jost, D. Merico, A. Mileo, P. Obermeier, O. Sabuncu, and T. Schaub, *Knowledge-intense stream reasoning*, Proceedings of the second workshop on logic-based interpretation of context: Modelling and applications (logic'11), 2011, pp. 7–8.

[20] A. Bogatarkan and E. Erdem, *Explanation generation for multi-modal multi-agent path finding with optimal resource utilization using answer set programming*, Theory and Practice of Logic Programming **20** (2020), no. 6, 974–989.

[21] J. Bomanson, T. Janhunen, and A. Weinzierl, *Enhancing lazy grounding with lazy normalization in answer-set programming*, Proceedings of the thirty-third national conference on artificial intelligence (aaai'19), 2019, pp. 2694–2702.

[22] G. Brewka, J. Delgrande, J. Romero, and T. Schaub, *Are preferences giving you a headache? — Take asprin!*, Proceedings of the seventh workshop on answer set programming and other computing paradigms (aspocp'14), 2014.

[23] ———, *asprin: Customizing answer set preferences without a headache*, Proceedings of the twenty-ninth national conference on artificial intelligence (aaai'15), 2015, pp. 1467–1474.

[24] G. Brewka, T. Eiter, and S. McIlraith (eds.), *Proceedings of the thirteenth international conference on principles of knowledge representation and reasoning (kr'12)*, AAAI Press, 2012.

[25] A. Bria, W. Faber, and N. Leone, *Normal form nested programs*, Fundamenta Informaticae **96** (2009), no. 3, 271–295.

[26] P. Cabalar, J. Fandinno, and B. Muñiz, *A system for explainable answer set programming*, Proceedings of the international conference on logic programming, 2020, pp. 124–136.

[27] P. Cabalar, R. Kaminski, P. Morkisch, and T. Schaub, *telingo = ASP + time*, Proceedings of the fifteenth international conference on logic programming and nonmonotonic reasoning (lpnmr'19), 2019, pp. 256–269.

[28] P. Cabalar, R. Kaminski, M. Ostrowski, and T. Schaub, *An ASP semantics for default reasoning with constraints*, Proceedings of the twenty-fifth international joint conference on artificial intelligence (ijcai'16), 2016, pp. 1015–1021.

[29] P. Cabalar, R. Kaminski, T. Schaub, and A. Schuhmann, *Temporal answer set programming on finite traces*, Theory and Practice of Logic Programming **18** (2018), no. 3-4, 406–420.

[30] P. Cabalar and T. Son (eds.), *Proceedings of the twelfth international conference on logic programming and nonmonotonic reasoning (lpnmr'13)*, Lecture Notes in Artificial Intelligence, vol. 8148, Springer-Verlag, 2013.

[31] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, M. Maratea, F. Ricca, and T. Schaub, *ASP-Core-2 input language format*, Theory and Practice of Logic Programming **20** (2020), no. 2, 294–309.

[32] F. Calimeri, M. Gebser, M. Maratea, and F. Ricca, *Design and results of the fifth answer set programming competition*, Artificial Intelligence **231** (2016), 151–181.

[33] F. Calimeri, G. Ianni, F. Pacenza, S. Perri, and J. Zangari, *Incremental answer set programming with overgrounding*, Theory and Practice of Logic Programming **19** (2019), no. 5-6, 957–973.

[34] F. Calimeri, G. Ianni, and F. Ricca, *3rd asp competition, file and language formats*, 2011.

[35] F. Calimeri, G. Ianni, and M. Truszczyński (eds.), *Proceedings of the thirteenth international conference on logic programming and nonmonotonic reasoning (lpnmr'15)*, Lecture Notes in Artificial Intelligence, vol. 9345, Springer-Verlag, 2015.

[36] M. Carro and A. King (eds.), *Technical communications of the thirty-second international conference on logic programming (iclp'16)*, OpenAccess Series in Informatics (OASIcs), vol. 52, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016.

[37] K. Clark, *Negation as failure*, Logic and data bases, 1978, pp. 293–322.

[38] K. Compton, A. Smith, and M. Mateas, *Anza island: Novel gameplay using ASP*, Proceedings of the the third workshop on procedural content generation in games, 2012, pp. 13:1–13:4.
[39] R. Conway, W. Maxwell, and L. Miller, *Theory of scheduling*, Addison-Wesley, 1967.
[40] S. Cotton and O. Maler, *Fast and flexible difference constraint propagation for DPLL(T)*, Proceedings of the ninth international conference on theory and applications of satisfiability testing (sat'06), 2006, pp. 170–183.
[41] B. Cuteri, C. Dodaro, F. Ricca, and P. Schüller, *Overcoming the grounding bottleneck due to constraints in ASP solving: Constraints become propagators*, Proceedings of the twenty-ninth international joint conference on artificial intelligence (ijcai'20), 2020, pp. 1688–1694.
[42] C. Dabral and C. Martens, *Generating explorable narrative spaces with answer set programming*, Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment **16** (October 2020), no. 1, 45–51.
[43] G. Dantzig, *Linear programming and extensions*, Princeton University Press, 1963.
[44] G. De Giacomo and M. Vardi, *Linear temporal logic and linear dynamic logic on finite traces*, Proceedings of the twenty-third international joint conference on artificial intelligence (ijcai'13), 2013, pp. 854–860.
[45] J. Delgrande and W. Faber (eds.), *Proceedings of the eleventh international conference on logic programming and nonmonotonic reasoning (lpnmr'11)*, Lecture Notes in Artificial Intelligence, vol. 6645, Springer-Verlag, 2011.
[46] M. Denecker, V. Marek, and M. Truszczyński, *Approximations, stable operators, well-founded fixpoints and applications in nonmonotonic reasoning*, Logic-based artificial intelligence, 2000, pp. 127–144.
[47] M. Denecker, N. Pelov, and M. Bruynooghe, *Ultimate well-founded and stable semantics for logic programs with aggregates*, Proceedings of the seventeenth international conference on logic programming (iclp'01), 2001, pp. 212–226.
[48] Y. Dimopoulos, M. Gebser, P. Lühne, J. Romero, and T. Schaub, *plasp 3: Towards effective ASP planning*, Theory and Practice of Logic Programming **19** (2018), no. 3, 477–504.
[49] Y. Dimopoulos, E. Kouppari, A. Philippou, and K. Psara, *Encoding reversing petri nets in answer set programming*, Reversible computation - 12th international conference, 2020, pp. 264–271.
[50] J. Dix, U. Furbach, and A. Nerode (eds.), *Proceedings of the fourth international conference on logic programming and nonmonotonic reasoning (lpnmr'97)*, Lecture Notes in Artificial Intelligence, vol. 1265, Springer-Verlag, 1997.
[51] A. Dovier and V. Santos Costa (eds.), *Technical communications of the twenty-eighth international conference on logic programming (iclp'12)*, Vol. 17, Leibniz International Proceedings in Informatics (LIPIcs), 2012.
[52] W. Dvorák, A. Rapberger, J. Wallner, and S. Woltran, *ASPARTIX-V19 - an answer-set programming based system for abstract argumentation*, Foundations of information and knowledge systems - 11th international symposium, 2020, pp. 79–89.
[53] T. Eiter, S. Germano, G. Ianni, T. Kaminski, C. Redl, P. Schüller, and A. Weinzierl, *The DLVHEX system*, Künstliche Intelligenz **32** (2018), no. 2-3, 187–189.
[54] T. Eiter, T. Kaminski, C. Redl, P. Schüller, and A. Weinzierl, *Answer set programming with external source access*, Reasoning web. semantic interoperability on the web - 13th international summer school 2017, 2017, pp. 204–275.
[55] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello, *A deductive system for nonmonotonic reasoning*, Proceedings of the fourth international conference on logic programming and nonmonotonic reasoning (lpnmr'97), 1997, pp. 363–374.
[56] E. Ellguth, M. Gebser, M. Gusowski, R. Kaminski, B. Kaufmann, S. Liske, T. Schaub, L. Schneidenbach, and B. Schnor, *A simple distributed conflict-driven answer set solver*, Proceedings of the tenth international conference on logic programming and nonmonotonic reasoning (lpnmr'09), 2009, pp. 490–495.
[57] E. Emerson, *Temporal and modal logic*, Handbook of theoretical computer science, 1990, pp. 995–1072.

[58] E. Erdem, M. Fidan, D. Manlove, and P. Prosser, *A general framework for stable roommates problems using answer set programming*, Theory and Practice of Logic Programming **20** (2020), no. 6, 911–925.

[59] E. Erdem and A. Herzig, *Solving gossip problems using answer set programming: An epistemic planning approach*, Proceedings of the international conference on logic programming, 2020, pp. 52–58.

[60] E. Erdem, F. Lin, and T. Schaub (eds.), *Proceedings of the tenth international conference on logic programming and nonmonotonic reasoning (lpnmr'09)*, Lecture Notes in Artificial Intelligence, vol. 5753, Springer-Verlag, 2009.

[61] F. Everardo, T. Janhunen, R. Kaminski, and T. Schaub, *The return of xorro*, Proceedings of the fifteenth international conference on logic programming and nonmonotonic reasoning (lpnmr'19), 2019, pp. 284–297.

[62] W. Faber, N. Leone, and G. Pfeifer, *Recursive aggregates in disjunctive logic programs: Semantics and complexity.*, Proceedings of the ninth european conference on logics in artificial intelligence (jelia'04), 2004, pp. 200–212.

[63] F. Fages, *Consistency of Clark's completion and the existence of stable models*, Journal of Methods of Logic in Computer Science **1** (1994), 51–60.

[64] P. Ferraris and V. Lifschitz, *Weight constraints as nested expressions*, Theory and Practice of Logic Programming **5** (2005), no. 1-2, 45–74.

[65] ———, *On the stable model semantics of first-order formulas with aggregates*, Proceedings of the thirteens international workshop on nonmonotonic reasoning (nmr'10), 2010.

[66] S. Gaggl, N. Manthey, A. Ronca, J. Wallner, and S. Woltran, *Improved answer-set programming encodings for abstract argumentation*, Theory and Practice of Logic Programming **15** (2015), no. 4-5, 434–448.

[67] M. Gebser, T. Grote, R. Kaminski, P. Obermeier, O. Sabuncu, and T. Schaub, *Stream reasoning with answer set programming: Preliminary report*, Proceedings of the thirteenth international conference on principles of knowledge representation and reasoning (kr'12), 2012, pp. 613–617.

[68] M. Gebser, T. Grote, R. Kaminski, and T. Schaub, *Reactive answer set programming*, Proceedings of the eleventh international conference on logic programming and nonmonotonic reasoning (lpnmr'11), 2011, pp. 54–66.

[69] M. Gebser, A. Harrison, R. Kaminski, V. Lifschitz, and T. Schaub, *Abstract Gringo*, Theory and Practice of Logic Programming **15** (2015), no. 4-5, 449–463.

[70] M. Gebser, T. Janhunen, H. Jost, R. Kaminski, and T. Schaub, *ASP solving for expanding universes*, Proceedings of the thirteenth international conference on logic programming and nonmonotonic reasoning (lpnmr'15), 2015, pp. 354–367.

[71] M. Gebser, T. Janhunen, R. Kaminski, T. Schaub, and S. Tasharrofi, *Writing declarative specifications for clauses*, Proceedings of the fifteenth european conference on logics in artificial intelligence (jelia'16), 2016, pp. 256–271.

[72] M. Gebser, R. Kaminski, B. Kaufmann, P. Lühne, P. Obermeier, M. Ostrowski, J. Romero, T. Schaub, S. Schellhorn, and P. Wanko, *The Potsdam answer set solving collection 5.0*, Künstliche Intelligenz **32** (2018), no. 2-3, 181–182.

[73] M. Gebser, R. Kaminski, B. Kaufmann, P. Lühne, J. Romero, and T. Schaub, *Answer set solving with generalized learned constraints*, Technical communications of the thirty-second international conference on logic programming (iclp'16), 2016, pp. 9:1–9:15.

[74] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and M. Schneider, *Potassco: The Potsdam answer set solving collection*, AI Communications **24** (2011), no. 2, 107–124.

[75] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele, *Engineering an incremental ASP solver*, Proceedings of the twenty-fourth international conference on logic programming (iclp'08), 2008, pp. 190–205.

[76] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and P. Wanko, *Theory solving made easy with clingo 5*, Technical communications of the thirty-second international conference on logic programming (iclp'16), 2016, pp. 2:1–2:15.

[77] M. Gebser, R. Kaminski, B. Kaufmann, J. Romero, and T. Schaub, *Progress in clasp series 3*, Proceedings of the thirteenth international conference on logic programming and nonmonotonic reasoning (lpnmr'15), 2015, pp. 368–383.

[78] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, *On the implementation of weight constraint rules in conflict-driven ASP solvers*, Proceedings of the twenty-fifth international conference on logic programming (iclp'09), 2009, pp. 250–264.

[79] ———, *Answer set solving in practice*, Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan and Claypool Publishers, 2012.

[80] ———, *Multi-shot ASP solving with clingo*, Theory and Practice of Logic Programming **19** (2019), no. 1, 27–82.

[81] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, M. Schneider, and S. Ziller, *A portfolio solver for answer set programming: Preliminary report*, Proceedings of the eleventh international conference on logic programming and nonmonotonic reasoning (lpnmr'11), 2011, pp. 352–357.

[82] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, and B. Schnor, *Cluster-based ASP solving with claspar*, Proceedings of the eleventh international conference on logic programming and nonmonotonic reasoning (lpnmr'11), 2011, pp. 364–369.

[83] M. Gebser, R. Kaminski, M. Knecht, and T. Schaub, *plasp: A prototype for PDDL-based planning in ASP*, Proceedings of the eleventh international conference on logic programming and nonmonotonic reasoning (lpnmr'11), 2011, pp. 358–363.

[84] M. Gebser, R. Kaminski, A. König, and T. Schaub, *Advances in gringo series 3*, Proceedings of the eleventh international conference on logic programming and nonmonotonic reasoning (lpnmr'11), 2011, pp. 345–351.

[85] M. Gebser, R. Kaminski, P. Obermeier, and T. Schaub, *Ricochet robots reloaded: A case-study in multi-shot ASP solving*, Advances in knowledge representation, logic programming, and abstract argumentation: Essays dedicated to Gerhard Brewka on the occasion of his 60th birthday, 2015, pp. 17–32.

[86] M. Gebser, R. Kaminski, M. Ostrowski, T. Schaub, and S. Thiele, *On the input language of ASP grounder gringo*, Proceedings of the tenth international conference on logic programming and nonmonotonic reasoning (lpnmr'09), 2009, pp. 502–508.

[87] M. Gebser, R. Kaminski, and T. Schaub, *aspcud: A Linux package configuration tool based on answer set programming*, Proceedings of the second international workshop on logics for component configuration (lococo'11), 2011, pp. 12–25.

[88] ———, *Complex optimization in answer set programming*, Theory and Practice of Logic Programming **11** (2011), no. 4-5, 821–839.

[89] ———, *Grounding recursive aggregates: Preliminary report*, Proceedings of the third workshop on grounding, transforming, and modularizing theories with variables (gttv'15), 2015.

[90] M. Gebser, B. Kaufmann, R. Otero, J. Romero, T. Schaub, and P. Wanko, *Domain-specific heuristics in answer set programming*, Proceedings of the twenty-seventh national conference on artificial intelligence (aaai'13), 2013, pp. 350–356.

[91] M. Gebser, B. Kaufmann, and T. Schaub, *Solution enumeration for projected Boolean search problems*, Proceedings of the sixth international conference on integration of ai and or techniques in constraint programming for combinatorial optimization problems (cpaior'09), 2009, pp. 71–86.

[92] M. Gebser, M. Maratea, and F. Ricca, *The sixth answer set programming competition*, Journal of Artificial Intelligence Research **60** (2017), 41–95.

[93] ———, *The seventh answer set programming competition: Design and results*, Theory and Practice of Logic Programming (2019). To appear.

[94] M. Gebser, P. Obermeier, and T. Schaub, *A system for interactive query answering with answer set programming*, Proceedings of the sixth workshop on answer set programming and other computing paradigms (aspocp'13), 2013.

[95] M. Gelfond and V. Lifschitz, *The stable model semantics for logic programming*, Proceedings of the fifth international conference and symposium of logic programming (iclp'88), 1988, pp. 1070–1080.

[96] ———, *Classical negation in logic programs and disjunctive databases*, New Generation Computing **9** (1991), 365–385.

[97] ———, *Action languages*, Electronic Transactions on Artificial Intelligence **3** (1998), no. 6, 193–210.
[98] M. Gelfond and Y. Zhang, *Vicious circle principle and logic programs with aggregates*, Theory and Practice of Logic Programming **14** (2014), no. 4-5, 587–601.
[99] A. Heyting, *Die formalen Regeln der intuitionistischen Logik*, Sitzungsberichte der preussischen akademie der wissenschaften, 1930, pp. 42–56.
[100] H. Hoos, R. Kaminski, M. Lindauer, and T. Schaub, *aspeed: Solver scheduling via answer set programming*, Theory and Practice of Logic Programming **15** (2015), no. 1, 117–142.
[101] H. Hoos, R. Kaminski, T. Schaub, and M. Schneider, *aspeed: ASP-based solver scheduling*, Technical communications of the twenty-eighth international conference on logic programming (iclp'12), 2012, pp. 176–187.
[102] H. Hoos, M. Lindauer, and T. Schaub, *claspfolio 2: Advances in algorithm selection for answer set programming*, Theory and Practice of Logic Programming **14** (2014), no. 4-5, 569–585.
[103] Y. Izmirlioglu and E. Erdem, *Reasoning about cardinal directions between 3-dimensional extended objects using answer set programming*, Theory and Practice of Logic Programming **20** (2020), no. 6, 942–957.
[104] T. Janhunen, R. Kaminski, M. Ostrowski, T. Schaub, S. Schellhorn, and P. Wanko, *Clingo goes linear constraints over reals and integers*, Theory and Practice of Logic Programming **17** (2017), no. 5-6, 872–888.
[105] T. Janhunen, G. Liu, and I. Niemelä, *Tight integration of non-ground answer set programming and satisfiability modulo theories*, Proceedings of the first workshop on grounding and transformation for theories with variables (gttv'11), 2011, pp. 1–13.
[106] R. Kaminski, J. Romero, T. Schaub, and P. Wanko, *How to build your own asp-based system?!*, Theory and Practice of Logic Programming **23** (2023), no. 1, 299–361.
[107] R. Kaminski and T. Schaub, *On the foundations of grounding in answer set programming*, Theory and Practice of Logic Programming (2023), 1–60.
[108] R. Kaminski, T. Schaub, A. Siegel, and S. Videla, *Minimal intervention strategies in logical signaling networks with answer set programming*, Theory and Practice of Logic Programming **13** (2013), no. 4-5, 675–690.
[109] R. Kaminski, T. Schaub, and P. Wanko, *A tutorial on hybrid answer set solving with clingo*, Proceedings of the thirteenth international summer school of the reasoning web, 2017, pp. 167–203.
[110] D. Kemp and P. Stuckey, *Semantics of logic programs with aggregates*, Proceedings of the 1991 international symposium on logic programming (islp'91), 1991, pp. 387–401.
[111] J. Lee and Y. Meng, *On reductive semantics of aggregates in answer set programming*, Proceedings of the tenth international conference on logic programming and nonmonotonic reasoning (lpnmr'09), 2009, pp. 182–195.
[112] C. Lefèvre, C. Béatrix, I. Stéphan, and L. Garcia, *ASPeRiX, a first-order forward chaining approach for answer set computing*, Theory and Practice of Logic Programming **17** (2017), no. 3, 266–310.
[113] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello, *The DLV system for knowledge representation and reasoning*, ACM Transactions on Computational Logic **7** (2006), no. 3, 499–562.
[114] Y. Lierler and B. Susman, *SMT-based constraint answer set solver EZSMT (system description)*, Technical communications of the thirty-second international conference on logic programming (iclp'16), 2016, pp. 1:1–1:15.
[115] V. Lifschitz, *What is answer set programming?*, Proceedings of the twenty-third national conference on artificial intelligence (aaai'08), 2008, pp. 1594–1597.
[116] F. Lin and X. Zhao, *On odd and even cycles in normal logic programs*, Proceedings of the nineteenth national conference on artificial intelligence, 2004, pp. 80–85.
[117] G. Liu, T. Janhunen, and I. Niemelä, *Answer set programming via mixed integer programming*, Proceedings of the thirteenth international conference on principles of knowledge representation and reasoning (kr'12), 2012, pp. 32–42.
[118] X. Liu and M. Truszczynski, *Aggregating conditionally lexicographic preferences using answer set programming solvers*, Algorithmic decision theory - third international conference, 2013, pp. 244–258.
[119] V. Marek and M. Truszczyński, *Autoepistemic logic*, Journal of the ACM **38** (1991), no. 3, 588–619.

[120] V. Nguyen, V. Stylianos, T. Son, and W. Yeoh, *Explainable planning using answer set programming*, Proceedings of the seventeenth international conference on principles of knowledge representation and reasoning (kr'21), 2020, pp. 662–666.

[121] I. Niemelä and P. Simons, *Smodels: An implementation of the stable model and well-founded semantics for normal logic programs*, Proceedings of the fourth international conference on logic programming and nonmonotonic reasoning (lpnmr'97), 1997, pp. 420–429.

[122] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry, *An A-Prolog decision support system for the space shuttle*, Proceedings of the third international symposium on practical aspects of declarative languages (padl'01), 2001, pp. 169–183.

[123] E. Oikarinen and T. Janhunen, *Modular equivalence for normal logic programs*, Proceedings of the seventeenth european conference on artificial intelligence (ecai'06), 2006, pp. 412–416.

[124] A. Dal Palù, A. Dovier, E. Pontelli, and G. Rossi, *GASP: Answer set programming with lazy grounding*, Fundamenta Informaticae **96** (2009), no. 3, 297–322.

[125] D. Pearce, *Equilibrium logic*, Annals of Mathematics and Artificial Intelligence **47** (2006), no. 1-2, 3–41.

[126] M. Razzaq, R. Kaminski, J. Romero, T. Schaub, J. Bourdon, and C. Guziolowski, *Computing diverse boolean networks from phosphoproteomic time series data*, Proceedings of the sixteenth international conference on computational methods in systems biology (cmsb'18), 2018, pp. 59–74.

[127] M. Rezvani, D. Rajaratnam, A. Ignjatovic, M. Pagnucco, and S. Jha, *Analyzing XACML policies using answer set programming*, International Journal of Information Security **18** (2019), no. 4, 465–479.

[128] E. Sandewall, *Features and fluents: the representation of knowledge about dynamical systems*, Vol. 1, Oxford University Press, New York, NY, USA, 1994.

[129] L. Schneidenbach, B. Schnor, M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, *Experiences running a parallel answer set solver on Blue Gene*, Proceedings of the sixteenth european pvm/mpi users' group meeting on recent advances in parallel virtual machine and message passing interface (pvm/mpi'09), 2009, pp. 64–72.

[130] C. Schultz, M. Bhatt, J. Suchan, and P. Walega, *Answer set programming modulo 'space-time'*, Rules and reasoning - second international joint conference, 2018, pp. 318–326.

[131] R. Schwitter, *Answer set programming via controlled natural language processing*, Controlled natural language - third international workshop, 2012, pp. 26–43.

[132] P. Simons, I. Niemelä, and T. Soininen, *Extending and implementing the stable model semantics*, Artificial Intelligence **138** (2002), no. 1-2, 181–234.

[133] A. Smith, E. Andersen, M. Mateas, and Z. Popovic, *A case study of expressively constrainable level design automation tools for a puzzle game*, International conference on the foundations of digital games, 2012, pp. 156–163.

[134] T. Son and E. Pontelli, *A constructive semantic characterization of aggregates in answer set programming*, Theory and Practice of Logic Programming **7** (2007), no. 3, 355–375.

[135] T. Syrjänen, *Lparse 1.0 user's manual*, 2001.

[136] ———, *Omega-restricted logic programs*, Proceedings of the sixth international conference on logic programming and nonmonotonic reasoning (lpnmr'01), 2001, pp. 267–279.

[137] E. Taillard, *Benchmarks for basic scheduling problems*, European Journal of Operational Research **64** (1993), no. 2, 278–285.

[138] E. Teppan. and G. Friedrich, *Heuristic constraint answer set programming for manufacturing problems* (I. Hatzilygeroudis and V. Palade, eds.), Springer-Verlag, 2018.

[139] M. Truszczyński, *Connecting first-order ASP and the logic FO(ID) through reducts*, Correct reasoning: Essays on logic-based AI in honour of Vladimir Lifschitz, 2012, pp. 543–559.

[140] ———, *An introduction to the stable and well-founded semantics of logic programs*, Declarative logic programming: Theory, systems, and applications, 2018, pp. 121–177.

[141] M. van Emden and R. Kowalski, *The semantics of predicate logic as a programming language*, Journal of the ACM **23** (1976), no. 4, 733–742.

[142] A. Van Gelder, *The alternating fixpoint of logic programs with negation*, Journal of Computer and System Sciences **47** (1993), 185–221.

[143] S. Videla, J. Saez-Rodriguez, C. Guziolowski, and A. Siegel, *caspo: a toolbox for automated reasoning on the response of logical signaling networks families*, Bioinformatics **33** (2017), no. 6, 947–950.

[144] A. Weinzierl, R. Taupe, and G. Friedrich, *Advancing lazy-grounding ASP solving techniques — restarts, phase saving, heuristics, and more*, Theory and Practice of Logic Programming **20** (2020), no. 5, 609–624.

[145] F. Wotawa, *On the use of answer set programming for model-based diagnosis*, Proceedings of the thirty-third international conference on industrial, engineering and other applications of applied intelligent systems (iea/aie'20), 2020, pp. 518–529.