Towards Efficient Online Reasoning About Actions

by

Joseph Babb

A Thesis Presented in Partial Fulfillment
of the Requirement for the Degree
Master of Science

Approved April 2014 by the
Graduate Supervisory Committee:

Joohyung Lee, Chair
Yann-Hang Lee
Chitta Baral

ARIZONA STATE UNIVERSITY

May 2014

ABSTRACT

Modeling dynamic systems is an interesting problem in Knowledge Representation (KR) due to their usefulness in reasoning about real-world environments. In order to effectively do this, a number of different formalisms have been considered ranging from low-level languages, such as Answer Set Programming (ASP), to high-level action languages, such as $\mathcal{C}+$ and $\mathcal{BC}$.

These languages show a lot of promise over many traditional approaches as they allow a developer to automate many tasks which require reasoning within dynamic environments in a succinct and elaboration tolerant manner. However, despite their strengths, they are still insufficient for modeling many systems, especially those of non-trivial scale or that require the ability to cope with exceptions which occur during execution, such as unexpected events or unintended consequences to actions which have been performed.

In order to address these challenges, a theoretical framework is created which focuses on improving the feasibility of applying KR techniques to such problems. The framework is centered on the action language $\mathcal{BC}+$, which integrates many of the strengths of existing KR formalisms, and provides the ability to perform efficient reasoning in an incremental fashion while handling exceptions which occur during execution. The result is a developer friendly formalism suitable for performing reasoning in an online environment.

Finally, the newly enhanced Cplus2ASP 2 is introduced, which provides a number of improvements over the original version. These improvements include implementing $\mathcal{BC}+$ among several additional languages, providing enhanced developer support, and exhibiting a significant performance increase over its predecessors and similar systems.

ACKNOWLEDGEMENTS

I would like to express my gratitude for my advisory, Dr. Joohyung Lee, as well as the rest of my thesis committee: Doctors Yann-Hang Lee and Chitta Baral, for taking their time and effort to assist me in achieving this accomplishment.

In addition, I would like to thank those that believed in me even when I did not: Mr. Ronald Wicks of American Fork Junior High, Mr. Frederick Fritz Fisher of Casa Grande Union High School, Professor Clark Vangilder of Central Arizona College, and last, but certainly not least, my grandmother Robbie McGalliard. Without the support of these individuals I doubt that I would have found myself where I am today.

Finally, I would like to thank my girlfriend Joan Miller, my parents Timothy and Lisa Wickert, and the rest of my family for their continued love and support.

The research presented herein was partially funded by:

Office of Secretary Defense-Test and Evaluation, Defense Wide / PE0601120D8Z
National Defense Education Program / BA-1, Basic Research
SMART Program Office, `http://www.asee.org/fellowships/smart`,
Grant Number N00244-09-1-0081.

National Science Foundation Grant No. {IIS-0916116, IIS-1036509, IIS-1319794}.

The Fulton Undergraduate Research Initiative (FURI).

TABLE OF CONTENTS

Page

LIST OF FIGURES vi

CHAPTER

1 INTRODUCTION 1

2 A BRIEF REVIEW OF BACKGROUND TOPICS 4

2.1 Answer Set Programming and the Stable Model Semantics 4

2.1.1 Partial Evaluation and Composition of Answer Set Programs 7

2.2 Action Languages 9

3 REVIEW OF ANSWER SET PROGRAMMING AND ACTION LANGUAGES 12

3.1 A Brief Note on First-Order Logic 12

3.2 The General Theory of Stable Models 15

3.3 A Note on Multi-Valued Formulas and the SM Semantics 19

3.4 Action Languages 20

3.4.1 The Action Language $\mathcal{B}$ 22

3.4.2 The Action Language $\mathcal{C}$ 26

3.4.3 The Action Language $\mathcal{C}+$ 30

3.4.4 The Action Language $\mathcal{BC}$ 33

4 THE ACTION LANGUAGE $\mathcal{BC}+$: INTEGRATING ASP, $\mathcal{C}+$, and $\mathcal{BC}$ .. 37

4.1 Defining the Action Language $\mathcal{BC}+$ 37

4.2 A Transition System Based Semantics 39

4.3 An ASP Based Semantics 41

4.4 Relation to Existing Formalisms 43

5 MODULAR AND ONLINE ANSWER SET PROGRAMS 46

5.1 The Symmetric Splitting Theorem 48

CHAPTER Page

5.2 The Module Theorem for Disjunctive Answer Set Programs 50
5.3 The Module Theorem for General Theory of Stable Models 53
5.4 Online Theories for Traditional Answer Set Programming 55
6 REVISITING LANGUAGE $\mathcal{BC}+$ WITH ONLINE EXECUTION 64
6.1 Online Theories for Propositional ASP Formulas 65
6.1.1 Reducing Online Propositional ASP Theories to Disjunctive Logic Theories 69
6.2 Defining the Online Action Language $\mathcal{BC}+$ 73
6.3 Transition Systems for $o\mathcal{BC}+$ 76
6.4 Incremental Assembly of Online $\mathcal{BC}+$ Descriptions 79
7 SYSTEM CPLUS2ASP 85
7.1 The Architecture of CPLUS2ASP 86
7.2 Modes of CPLUS2ASP 88
7.3 Running Modes of CPLUS2ASP 89
7.4 The Input Languages of CPLUS2ASP 90
7.4.1 Shared Syntax 90
7.4.2 Multi-Valued (Propositional) Formulas (MVPF) 104
7.4.3 The Action Language $\mathcal{C}+$ 105
7.4.4 The Action Language $\mathcal{BC}$ 110
7.4.5 The Action Language $\mathcal{BC}+$ 111
7.5 Running CPLUS2ASP 113
7.5.1 Using the Command-Line Mode 113
7.5.2 Using the Interactive Mode 116
7.6 Experiments 118

CHAPTER Page

8 CONCLUSION 122

REFERENCES 124

APPENDIX

A REFERENCED ACTION DESCRIPTIONS 128

A.1 The Action Language $\mathcal{B}$ 129

A.2 The Action Language $\mathcal{C}$ 130

A.3 The Action Language $\mathcal{C}+$ 130

A.4 The Action Language $\mathcal{BC}$ 133

A.5 The Action Language $\mathcal{BC}+$ 134

B PROOFS OF STATEMENTS 137

B.1 Proposition 2 138

B.2 Proposition 3 141

B.3 Proposition 4 141

B.4 Theorem 3 141

B.5 Proposition 7 144

B.6 Proposition 8 144

B.7 Lemma 1 146

B.8 Proposition 9 152

B.9 Proposition 10 153

B.10 Proposition 11 159

LIST OF FIGURES

Figure Page

2.1 A basic toggle switch in ASP 5

3.1 The Pendulum Problem in $\mathcal{B}$ 25

3.2 The Publishing Problem in $\mathcal{B}$ 26

3.3 Attempting to Express the Many Switches Problem in $\mathcal{C}$ 29

3.4 The Publishing Problem in $\mathcal{C}$ 30

3.5 The Pendulum Problem in $\mathcal{BC}$ 35

4.1 Toggle Switch Domain in $\mathcal{BC}+$. 39

4.2 The transition system of $\mathcal{D}_{\text{switch}}$ 40

5.1 Module Instantiation of a Simple Traditional ASP Program 56

5.2 Precedence graph of component formulas 60

6.1 Hierarchy of symbols within a $\mathcal{BC}+$ signature 74

6.2 Online Faulty Switch Elaboration in $o\mathcal{BC}+$. 76

6.3 A Partial Transition System of $\mathcal{D}^{o\mathcal{BC}+}_{\text{switch}}$ 78

7.1 CPLUS2ASP v2 System Architecture 86

7.2 8-queens in MVPF 105

7.3 The Pendulum Problem in $\mathcal{C}+$ 109

7.4 Benchmarking Results 119

7.5 Ferryman 120/4 Long Horizon Analysis 120

A.1 The Light Switch Problem in $\mathcal{B}$ 129

A.2 The Many Switch Problem in $\mathcal{B}$ 129

A.3 The Light Switch Problem in $\mathcal{C}$ 130

A.4 The Pendulum Problem in $\mathcal{C}$ 130

A.5 The Light Switch Problem in $\mathcal{C}+$ 131

A.6 The Pendulum Problem in $\mathcal{C}+$ 131

Figure Page

A.7 The Publishing Problem in $\mathcal{C}+$ 132
A.8 The Light Switch Problem in $\mathcal{BC}$ 133
A.9 The Many Switch Problem in $\mathcal{BC}$ 133
A.10 The Publishing Problem in $\mathcal{BC}$ 134
A.11 The Many Switch Problem in $\mathcal{BC}+$ 135
A.12 The Pendulum Problem in $\mathcal{BC}+$ 135
A.13 The Publishing Problem in $\mathcal{BC}+$ 136

Chapter 1

INTRODUCTION

Dynamic systems are systems which evolve over time due to actions performed by agents, such as pushing a ball, or non-inertial changes in the environment, such as the ball continuing to roll down a slope. Naturally, dynamic systems make up a large portion of systems considered in the field of Knowledge Representation (KR) due to their usefulness in modeling the evolution of real-world environments over time.

Many KR formalisms have been considered for modeling dynamic systems with varying strengths and weaknesses. While low level languages, such as Answer Set Programming (ASP) (Gelfond and Lifschitz (1988)), provide an extremely expressive environment capable of representing many interesting properties of systems, they are seen colloquially as "logical assembly languages" which have a steep learning curve and provide little-to-no opportunity for developmental support. Meanwhile, high-level *action languages* (Gelfond and Lifschitz (1998)) provide a more structured representation of dynamic systems, leading to a succinct and intuitive syntax and semantics and creating a dramatically more developer friendly formalism. However, this structure often comes at the cost of expressivity.

Using these languages, a developer is able to automate many tasks which require non-trivial reasoning within dynamic environments in a succinct and elaboration tolerant manner. This is accomplished by allowing the developer to specify the properties of the system and pushing the burden of reasoning a bout the system from the developer to an automated solver.

These formalisms have, traditionally, been considered in an offline environment where agents within each system are assumed to have complete knowledge of the

system's current state and the results of executing each action within the system. This is a very strong assumption in practice as real-world environments do not always behave in a predictable manner. Recently, attempts have been made to create fault-tolerant reasoning and execution systems at a meta-level by throwing out the previous results and restarting from scratch when an exception occurs (Erdem *et al.* (2011)). Unfortunately, this approach is not sufficient, especially when considering large-scale or time-sensitive problems, due to the intractable nature of modeling and reasoning with respect to dynamic systems.

How can we provide a knowledge representation system suitable for dynamic reasoning which is easily accessible to developers and users while providing the scalability and flexibility required to be feasible in a greater range of problems?

In this thesis we consider these challenges and propose an integrated approach to mitigate them. Our specific contributes are as follows:

- we design the online action language $\mathcal{BC}+$, a high-level language whose offline specialization generalizes several existing action languages and closes the expressivity gap between these languages and Answer Set Programming;
- we provide a generalization of the Module Theorem (Oikarinen and Janhunen (2006); Janhunen *et al.* (2009)) to allow for the modular decomposition and evaluation of first-order formulas under the Stable Model Semantics (Ferraris *et al.* (2009a));
- we then apply this generalized module theorem to encapsulate online $\mathcal{BC}+$ within the online ASP theories behind iClingo (Gebser *et al.* (2008)) and oClingo (Gebser *et al.* (2011a)), which yields a unified framework for modeling dynamic systems and providing fault tolerant plan execution in an efficient manner;

- finally, we show the efficacy of our approach by supplying a new version of the system CPLUS2ASP (Casolary and Lee (2011)) which enhances the original version in a number of ways, including providing an implementation of online $\mathcal{BC}+$.

The document is structured as follows: in Chapter 2 we give a brief overview of related research; in Chapter 3 we provide an in-depth technical review of Answer Set Programming and several action languages of interest; following this, Chapter 4 defines a new language $\mathcal{BC}+$ designed to take advantage of the strengths of each of these existing formalisms; Chapter 5 discusses and extends the existing efforts for evaluating ASP programs in a compositional manner; afterward, Chapter 6 uses these results to define an online extension of $\mathcal{BC}+$ to allow for more efficient fault tolerant reasoning; finally, Chapter 7 showcases system CPLUS2ASP 2.0, which has many improvements over its predecessor including support for $\mathcal{BC}+$.

Chapter 2

A BRIEF REVIEW OF BACKGROUND TOPICS

In this chapter, we briefly review the history of the works which our contributions are built from. For brevity we exclude the technical details of each work and instead summarize the results. Instead, technical reviews of select topics are included in later chapters.

2.1 Answer Set Programming and the Stable Model Semantics

Answer Set Programming is a logical programming language which is well suited for solving NP-complete combinatorial problems. Its syntax consists of *rules* which are evaluated based on non-monotonic fixpoint semantics. A rule in traditional Answer Set Programming is an expression of the form

$$a_0 \leftarrow a_1, \ldots, a_n, \text{not } a_{n+1}, \ldots, \text{not } a_m, \tag{2.1}$$

where each a_i $(0 \leq i \leq m)$ is a Boolean *atomic proposition*, also known as an *atom*, or the 0-place connected $\bot$. We call a_0 the *head* of the rule, the remainder of the rule is the *body*.

Each rule (2.1) may also be viewed syntactically as the propositional logic implication

$$\neg a_m \wedge \cdots \wedge \neg a_{n+1} \wedge a_n \wedge \cdots \wedge a_1 \rightarrow a_0. \tag{2.2}$$

Intuitively, each ASP rule is read as "a_0 holds if each a_i $(1 \leq i \leq n)$ has been shown to hold and it is consistent to assume that each a_j $(n+1 \leq j \leq m)$ does not hold. This notion is captured by checking if each potential solution against a fixpoint construction, known as the *reduct*.

$\sim on0 \leftarrow \text{not } on0.$	$D_{\text{switch},1}$
$on0 \leftarrow \text{not } \sim on0.$	$D_{\text{switch},2}$
$on1 \leftarrow on0, \text{not } flip.$	$D_{\text{switch},3}$
$on1 \leftarrow flip, \text{not } on0.$	$D_{\text{switch},4}$
$\sim flip \leftarrow \text{not } flip.$	$D_{\text{switch},5}$
$flip \leftarrow \text{not } \sim flip.$	$D_{\text{switch},6}$
$\bot \leftarrow \text{not } on1.$	$D_{\text{switch},7}$

Figure 2.1: A basic toggle switch in ASP

Example 1 *As an example, consider the ASP program shown in Figure 2.1 which describes a simple toggle switch. Intuitively, on0 and on1 characterize the initial and final states of the switch, respectively, and flip describes the act of toggling the switch.*

Rules 1 and 2 use an auxiliary atom, ∼on0, in order to allow for the initial state of the switch to be chose arbitrarily. Intuitively, these rules interact to provide the program with a choice as to which of on0 and ∼on0 are asserted. Rules 5 and 6 do the same for flip, allowing it to be executed arbitrarily.

Rules 3 and 4 provide the effect of executing toggle on the switch; specifically, rule 3 states that if toggle is not executed and the switch was on it should stay that way, while rule 4 states that if toggle is executed and the switch was off it should turn on. It is not necessary to explicitly state when the switch turns off as the semantics of ASP cause on1 to be false when not forced to be true.

Finally, rule 7 provides a constraint on the final state of the toggle switch by intuitively stating that it should not be the case that on1 is false.

This program has two solutions, or answer sets:

$$\{\sim on0, flip, on1\}, \text{ and } \{on0, \sim flip, on1\}.$$

In the first solution, the switch is initially off and the agent flips it on. Meanwhile, in the second solution the switch is initially on and the agent performs no action which causes the switch to remain on by inertia.

More recently, ASP has been generalized to allow for rules of the form

$$a_0 | \ldots | a_k \leftarrow a_{k+1}, \ldots, a_n, \text{not } a_{n+1}, \ldots, \text{not } a_m, \tag{2.3}$$

where there is a disjunction of atoms in the consequent of each rule (Gelfond and Lifschitz (1991)), to allow for the full syntax of nested propositional formulas (Ferraris (2005)), to be applicable to first-order formulas with intensional predicates under the Stable Model (SM) Semantics (Ferraris *et al.* (2009a)), allow for more expressive generalized quantifiers (Lee and Meng (2012)), and allow for intensional functions (Bartholomew and Lee (2012)). These generalizations have vastly increased the potential expressivity and succinctness of ASP programs as well as the academic community's understanding of their properties.

Answer Set Programming is a powerful formalism and has garnered a lot of attention since its inception. Since then, several efficient implementations for ASP solvers, including Gringo (Gebser *et al.* (2007b, 2009, 2011b)) / Clasp (Gebser *et al.* (2007a)) as well as the DLV system (Leone and et al. (2005); Leone *et al.* (2006a,b)) have been created. The presence of these systems, as well as the ability to succinctly encode NP-complete reasoning problems, has lead to the exploration of numerous applications.

Despite this, ASP has a number of drawbacks which must be addressed before the formalism can see a wider spread adoption. The syntax of ASP is low-level and is often thought of colloquially as a "logical assembly language", rather than a fully developed programming language. This, combined with the difficulty in mastering its semantics and the lack of development tools, provides for a steep learning curve for

creating and debugging ASP programs. In addition to this, the intractable nature of solving ASP programs restricts the size of the problems which ASP must be applied to.

2.1.1 Partial Evaluation and Composition of Answer Set Programs

The intractability of Answer Set Programming has proven to be a difficult problem to work around. There are certain classes of ASP programs which can be solved efficiently, such as Horn programs, which contain only ASP rules of the form (2.1) where $n = m$ (i.e. contains no occurrences of "not a" where a is an atom). Unfortunately, these classes of programs are not capable of expressing many interesting properties. For example, the toggling behavior shown in Example 1 is not expressible using a Horn program due to their inability to represent choice.

An alternative approach to addressing the scalability issues with ASP is to divide the problem into parts and solve each one independently. The Splitting Theorem (Lifschitz and Turner (1994); Ferraris *et al.* (2009a)) is one method capable of doing just this. It provides a method of decomposing a first-order formula under the Stable Model Semantics into smaller formulas, of the same signature, which are independent of one another. Determining if a specific interpretation is a solution to the monolithic formula can then be done by checking if it is a solution for each of the sub-formulas.

The Module Theorem (Oikarinen and Janhunen (2006); Janhunen *et al.* (2007)) takes the opposite approach. It views a program as being composed of a series of small disjunctive ASP *modules*, each with their own signature and input/output interface, and provides a formal method of composing the solutions of each sub-module to generate the solutions of the whole program.

While the Splitting Theorem can be applied to decompose an ASP program for the purposes of solution checking, it cannot be directly applied in order to generate solu-

tions of the larger problem given its components. Although this problem is addressed in the Module Theorem, the Module Theorem is not applicable to the general case of first-order formulas under the Stable Model Semantics, which provides for many useful features including arbitrary nested formulas, aggregates used for cardinality counting and finding the minimum and maximum values in a set, and quantifiers.

Despite this limitation, the Module Theorem has been applied in practice in order to accelerate solving problems with ASP which involve an iterative deepening search or those in which complete knowledge cannot be assumed ahead of time.

Typically, an iterative deepening search, such as finding the minimum length plan to achieve a goal, consists of restarting the grounding[1]/solving process repeatedly while varying the maximum step parameter. This process involves the entire program being re-grounded and re-solved from scratch a number of times, resulting in a lot of repeated work. This problem is handled by the system iClingo (Gebser *et al.* (2008)).

iClingo applies the Module Theorem in order to allow programs to be incrementally instantiated from a template incremental ASP program and composed together during an iterative solving process. In practice, this results in a significant speedup to problems which lend themselves to an iterative deepening search as re-grounding previous steps is avoided and heuristics may be saved and used across solver calls.

Similarly, accounting for incomplete knowledge in a system has traditionally been handled by restarting the grounding/solving process with any new information that has been acquired during execution. The system oClingo (Gebser *et al.* (2011a,

[1] In practice ASP programs are specified in a pseudo-first-order manner by using schematic variables in each rule which are replaced with all possible values the variable can take in a process called grounding. The result is a propositional ASP program which can be reasoned over.

2012)), which builds on the iClingo system prevents this by allowing for simple online ASP modules containing this information to be dynamically added to the incremental program.

iClingo and oClingo have shown promise in enabling the use of ASP in systems of non-trivial scale. The primary purpose of both systems is to prevent multiple unnecessary restarts of the grounding/solving process which cause much of the work to be redone. However, the conditions which they impose on programs compound the complexity of developing ASP programs by placing additional burdens on the user to check the correctness of their programs.

2.2 Action Languages

Action languages are "formal models of parts of the natural language that are used to talk about the effects of actions" (Gelfond and Lifschitz (1998)). Essentially, they provide a formal semantics for some fragment of natural language which may then be reasoned about. This distinguishes them from other logical formalisms, such as ASP, which model formal logics. The additional structure provided by action languages results in them being more accessible to the community at large; however, as a side effect, they also tend to be more restricted in terms of expressivity than their lower-level counterparts.

Gelfond and Lifschitz (1998) provide definitions for two particular action languages of interest, $\mathcal{B}$ and $\mathcal{C}$ (Giunchiglia and Lifschitz (1998)), and give their semantics in terms of transition systems which can be generated from provided *action descriptions*, or programs, within each language. Although the two languages share many commonalities, neither of them subsumes the other in terms of expressivity (Gelfond and Lifschitz (2012)). $\mathcal{C}$ was initially implemented in the Causal Calculator (CCalc) (McCain (1997)) via a reduction into propositional satisfiability checking.

The action language $\mathcal{C}$ was enhanced by Giunchiglia *et al.* (2004) to create the action language $\mathcal{C}+$. $\mathcal{C}+$ enhances $\mathcal{C}$ in a number of ways including providing for non-exogenous actions and complex dependencies between atoms. The semantics of $\mathcal{C}+$ were formalized in terms of non-monotonic causal logic (Giunchiglia *et al.* (2004)) as well as transition systems. The definite propositional fragment of $\mathcal{C}+$ was implemented in Causal Calculator version 2 (CCalc 2) (Lee (2005)).

Eventually, a translation from the languages $\mathcal{C}$ and $\mathcal{B}$ to ASP was proposed by Lifschitz and Turner (1999) and Son *et al.* (2002), respectively, was implemented in the system coala (Gebser *et al.* (2010)). Shortly after, Casolary and Lee (2011) leveraged a translation from definite causal logic to formulas under the Stable Model Semantics (Ferraris (2007); Ferraris *et al.* (2012)), which could then be further reduced into traditional ASP when the domain of consideration is fixed and finite (Lee and Palla (2009)), in order to reduce $\mathcal{C}+$ to ASP in the system Cplus2ASP. Both of these systems show a significant performance increase over their predecessors as they are able to leverage highly-optimized ASP solvers (Casolary and Lee (2011)).

Finally, many of the strengths, both syntactic and semantic, of $\mathcal{B}$ and $\mathcal{C}$ by Gelfond and Lifschitz (1998), were built on in order to create a the language $\mathcal{BC}$ by Lee *et al.* (2013). $\mathcal{BC}$ generalizes $\mathcal{B}$ with a number of features found in $\mathcal{C}$, which provides a significant improvement in the flexibility of the language. Lee *et al.* (2013) described the semantics of $\mathcal{BC}$ in terms of a reduction to several variants of ASP.

Action languages have a major advantage over Answer Set Programming in that they do not fall prey to the cryptic nature of ASP. In addition, the structure imposed by many action languages allows for systems to provide additional support for developers via techniques such as enhanced static semantic checking as well as transition system visualization tools.

However, finding the balance between structured syntax and expressivity is difficult. As we'll see in Chapter 3, many of these action languages are too restrictive to express many concepts easily.

Chapter 3

REVIEW OF ANSWER SET PROGRAMMING AND ACTION LANGUAGES

In this chapter we provide a technical review for the formalisms which our work is based on. We first begin by briefly reviewing the syntax and semantics of propositional and first-order logic. Following this, we review the Stable Model Semantics proposed by Ferraris *et al.* (2011), and, as a special case, the semantics of propositional ASP. Finally, we introduce and compare the action languages $\mathcal{B}$, $\mathcal{C}$, $\mathcal{C}+$ and $\mathcal{BC}$ and describe their semantics in terms of a reduction into multi-valued formulas under the stable model semantics.

In the following chapters we will use these formalism as a starting point to define the action language $\mathcal{BC}+$, which seeks to take advantage of the strengths of each of them.

3.1 A Brief Note on First-Order Logic

A first-order signature σ is a set of predicate and function symbols, each with an associated non-negative *arity*. We will commonly refer to predicates and function symbols with 0 arity as *atoms* and *object symbols*, respectively.

In addition to a signature, we assume the presence of an infinite set of variable symbols $\{x_1, x_2, \dots\}$.

A first-order formula of signature σ is constructed from predicate and formula symbols in σ, the nullary connective $\bot$, binary connectives $\wedge$, $\vee$, and $\rightarrow$, variables, and quantifiers $\forall$ and $\exists$.

A first-order formula is defined recursively as follows:

- a variable is a term;

- given a function symbol $f \in \sigma$ of arity n and terms $t_1, \ldots, t_n$, $f(t_1, \ldots, t_n)$ is a term;
- $\bot$ is a first-order formula;
- given a predicate symbol $p \in \sigma$ of arity n and terms $t_1, \ldots, t_n$, $p(t_1, \ldots, t_n)$ is a first-order formula;
- given first-order formulas F and G, $F \odot G$ is a first-order formula where $\odot$ is a binary connective; and,
- given a first-order formula F, $\forall x F(x)$ and $\exists x F(x)$ are first-order formulas.

Following Ferraris *et al.* (2011), we view $\neg F$ and $\top$ as shorthand for $F \rightarrow \bot$ and $\neg\bot$, respectively. Finally, we call each given a formula F of the form

$$F_1 \wedge F_2 \wedge \cdots \wedge F_n,$$

we call each F_i $(1 \leq i \leq n)$ a *rule* of F.

As an example, given a signature $\sigma_{\text{EX}} = \{p, q, r, a, b\}$ where p, q, r are unary predicate symbols and a, b are object symbols,

$$p(a) \wedge q(b) \wedge \forall x((\neg q(x) \wedge p(x)) \rightarrow r(x)) \tag{3.1}$$

is a first-order formula, whereas "$p(a) \wedge a$" and "$p(q(b))$" are not. The rules of (3.1) are $p(a)$, $q(b)$ and $\forall x((\neg q(x) \wedge p(x)) \rightarrow r(x))$.

The semantics of a first-order logic are stated in terms of the satisfaction of *first-order interpretations*. A first order interpretation I consists of a universe of symbols $|I|$ and a function c^I for each $c \in \sigma$ of arity n such that $c^I : |I|^n \mapsto |I|$ if c is a function symbol and $c^I : |I|^n \mapsto \{\mathtt{t}, \mathtt{f}\}$ otherwise.

We say that a formula F is a sentence if, for every occurrence of a variable x within F, the occurrence is within some subformula $QxG(x)$ such that $Q \in \{\forall, \exists\}$. Additionally, we say term or formula is *ground* if it does not contain any variables.

Given an interpretation I of signature σ, by σ^I we denote the signature obtained by adding a new object symbol ξ^* to σ for each $\xi \in |I|$. Additionally, it is assumed that I is extended with each such ξ^* by letting $\xi^{*I} = \xi$.

As an example, given the previous signature σ_{EX} and an interpretation such that $|I|$ is $\{0, 1, 2, 3\}$, σ^I_{EX} is $\{p, q, r, a, b, 0^*, 1^*, 2^*, 3^*\}$.

Given a first order sentence F and first-order interpretation I of the same signature σ, the evaluation of F with respect to I (denoted F^I) is defined recursively as:

- $\bot^I$ is $\mathtt{f}$;
- $c(t_1, \ldots, t_n)^I$ is $c^I(t^I_1, \ldots, t^I_n)$ for any constant symbol c of arity n and ground terms $t_1, \ldots, t_n$;
- $(F \wedge\ G)^I$ is F^I and G^I;
- $(F \vee\ G)^I$ is F^I or G^I;
- $(F \rightarrow\ G)^I$ is F^I implies G^I;[1]
- $\forall x F(x)$ is $\mathtt{t}$ iff, for each $\xi \in |I|$, $F(\xi^\star)^I$ is $\mathtt{t}$;
- $\exists x F(x)$ is $\mathtt{t}$ iff, for some $\xi \in |I|$, $F(\xi^\star)^I$ is $\mathtt{t}$.

We say an interpretation I satisfies a first-order sentence F (denoted $I \models F$) if $F^I = \mathtt{t}$.

Example 2 *As an example, Let I and J be first-order interpretations of the signature σ_{EX} such that*

$$|I| = \{a, b\}, \quad p^I = \{a\},^2 \quad q^I = \{b\}, \quad r^I = \{a\}, \quad a^I = a, \text{ and } \quad b^I = b.$$

and

$$|J| = \{1, 2, 3\}, \quad p^J = \{2, 3\}, \quad q^J = \{1, 2\}, \quad r^J = \emptyset, \quad a^J = 2, \text{ and } \quad b^J = 2.$$

[1] Equivalently, G^I or not F^I.

[2] Here we identify each function with the set of tuples that it maps to $\mathtt{t}$.

I satisfies (3.1)*, however J does not as $\neg q(3^*) \wedge p(3^*) \rightarrow r(3^*)$ is not satisfied by J. However, if were to let $r^J = \{3\}$ then J would satisfy* (3.1)*.*

Finally, in the event the signature is propositional (i.e. only contains atoms), we sometimes will identify an interpretation I with the set of atoms mapped to `t`.

Second-order logic adopts a similar structure and definition, except that it allows for quantification over *predicate variables*, which range over all possible functions from a power set of Cartesian products of the universe to `t` and `f`.

3.2 The General Theory of Stable Models

The Stable Model (SM) Semantics were proposed by Ferraris *et al.* (2011) as a first-order extension of Answer Set Programming. Rather than define their semantics using a fixpoint construct as has been done traditionally for ASP, they instead introduce the SM operator, which characterizes the *stable models* of a first-order formula F using a second order formula constructed from F with respect to a set of *intensional predicates* fully characterized by F.

Moving forward, we assume that the underlying signature σ has a finite number of predicate symbols.

Given lists $\mathbf{p}$ and $\mathbf{u}$ of distinct predicate symbols $p_1, \ldots, p_n$ and predicate variables $u_1, \ldots, u_n$, respectively. We define $\mathbf{u} \leq \mathbf{p}$ to be the conjunction of formulas

$$\forall \mathbf{x}_i (u_i(\mathbf{x}_i) \rightarrow p_i(\mathbf{x}_i))$$

such that $\mathbf{x}_i$ is a list of object variables of the same length as the arity of p_i. Additionally, we define $\mathbf{u} < \mathbf{p}$ to be the formula $(\mathbf{u} \leq \mathbf{p}) \wedge \neg(\mathbf{p} \leq \mathbf{u})$.[3]

[3] The definition of $\mathbf{p} \leq \mathbf{u}$ mirrors that of $\mathbf{u} \leq \mathbf{p}$.

Example 3 *For example, if p and q are unary predicate constants then $(u, v) < (p, q)$ is*

$$\forall x(u(x) \to p(x)) \wedge \forall x(v(x) \to q(x)) \wedge \neg\Big(\forall x(p(x) \to u(x)) \wedge \forall x(q(x) \to v(x))\Big).$$

For any first-order formula F, Ferraris *et al.* (2011) define $\mathrm{SM}[F; \mathbf{p}]$ to be

$$F \wedge \neg\exists \mathbf{u}((\mathbf{u} < \mathbf{p}) \wedge F^*(\mathbf{u})),$$

where $F^*(\mathbf{u})$ is defined recursively as follows:

- $p_i(\mathbf{t})^* = u_i(\mathbf{t})$ for any list $\mathbf{t}$ of terms;
- $F^* = F$ for any atomic formula F (including $\bot$ and equality) that does not contain members of $\mathbf{p}$;
- $(F \wedge G)^* = F^* \wedge G^*$;
- $(F \vee G)^* = F^* \vee G^*$;
- $(F \to G)^* = (F^* \to G^*) \wedge (F \to G)$;
- $(\forall x F)^* = \forall x F^*$;
- $(\exists x F)^* = \exists x F^*$.

When F is a sentence, the models of $\mathrm{SM}[F; \mathbf{p}]$ are called the $\mathbf{p}$-*stable* models of F. Intuitively, they are the models of F that are "stable" on $\mathbf{p}$. We will often simply write $\mathrm{SM}[F]$ in place of $\mathrm{SM}[F; \mathbf{p}]$ when $\mathbf{p}$ is the list of all predicate constants within the signature.

Proposition 1 (ASP within the SM Semantics) *(Ferraris* et al. *(2011)) Given an Answer Set Program F, the stable models of* $\mathrm{SM}[F]$ *correspond to the answer sets of F.*

Example 4 *As an example, consider the program F shown in Figure 2.1, which may also be viewed as the propositional logic formula*

$$\begin{aligned}
&\neg on \rightarrow {\sim}on \\
\wedge\ &\neg on \rightarrow {\sim}on \\
\wedge\ &\neg flip \wedge on0 \rightarrow on1 \\
\wedge\ &\neg on0 \wedge flip \rightarrow on1 \\
\wedge\ &\neg flip \rightarrow {\sim}flip \\
\wedge\ &\neg{\sim}flip \rightarrow flip \\
\wedge\ &\neg on1 \rightarrow \top
\end{aligned}$$

of signature $\sigma = \{on0, {\sim}on0, flip, {\sim}flip, on1\}$.

$F^*(\mathbf{u})$ *is*

$$\begin{aligned}
&\neg on0 \rightarrow {\sim}on0 && \wedge \neg on0 \wedge \neg u_{on0} \rightarrow u_{{\sim}on0} \\
\wedge\ &\neg{\sim}on0 \rightarrow on0 && \wedge \neg{\sim}on0 \wedge \neg u_{{\sim}on0} \rightarrow u_{on0} \\
\wedge\ &\neg flip \wedge on0 \rightarrow on1 && \wedge \neg flip \wedge \neg u_{flip} \wedge u_{on0} \rightarrow u_{on1} \\
\wedge\ &\neg on0 \wedge flip \rightarrow on1 && \wedge \neg on0 \wedge \neg u_{on0} \wedge u_{flip} \rightarrow u_{on1} \\
\wedge\ &\neg flip \rightarrow {\sim}flip && \wedge \neg flip \wedge \neg u_{flip} \rightarrow u_{{\sim}flip} \\
\wedge\ &\neg{\sim}flip \rightarrow flip && \wedge \neg{\sim}flip \wedge \neg u_{{\sim}flip} \rightarrow u_{flip} \\
\wedge\ &\neg on1 \rightarrow \top && \wedge \neg on1 \wedge \neg u_{on1} \rightarrow \top
\end{aligned}$$

where each u_c is a predicate variable corresponding to c. Note that the rules on the left correspond exactly to F.

As $u_c \leq c$ for each $c \in \sigma$, it holds that $\neg c \rightarrow \neg u_c$. Due to this, we can replace all occurrences of $\neg c \wedge \neg u_c$ with $\neg c$ in each rule body to obtain $F \wedge F^{*\prime}(\mathbf{u})$ where $F^{*\prime}(\mathbf{u})$ is

$$
\begin{aligned}
&\neg on0 \rightarrow u_{\sim on0} \\
\wedge\ &\neg\sim on0 \rightarrow u_{on0} \\
\wedge\ &\neg flip \wedge u_{on0} \rightarrow u_{on1} \\
\wedge\ &\neg on0 \wedge u_{flip} \rightarrow u_{on1} \\
\wedge\ &\neg flip \rightarrow u_{\sim flip} \\
\wedge\ &\neg\sim flip \rightarrow u_{flip} \\
\wedge\ &\neg on1 \rightarrow \top
\end{aligned}
$$

It then follows that an interpretation I is a stable model of F if it is a model of

$$F \wedge \neg\exists \mathbf{u}((\mathbf{u} < \mathbf{p}) \wedge F \wedge F^{*\prime}(\mathbf{u})),$$

or, equivalently,

$$F \wedge \neg\exists \mathbf{u}((\mathbf{u} < \mathbf{p}) \wedge F^{*\prime}(\mathbf{u}))$$

If $I = \{\sim on0, flip, on1\}$ it is clear that $I \models F$ and that $u_{\sim on0}$, u_{flip}, u_{on1} must be asserted in order to satisfy $F^{*\prime}(\mathbf{u})$. It then follows that I is a stable model of F. Similarly, it can be shown that $\{on0, \sim flip, on1\}$ is also a stable model of F.

Furthermore, $J = \{on0, \sim flip, on1\}$ also satisfies F. However, it is easy to see that asserting u_{on0} and $u_{\sim flip}$ is sufficient in order to satisfy $F^{*\prime}(\mathbf{u})$, which violates the condition that $\neg\exists \mathbf{u}((\mathbf{u} < \mathbf{p}) \wedge F^{*\prime}(\mathbf{u}))$.

This agrees with our analysis of the program in Example 1.

Using the SM semantics it is trivial to generalize answer set programming to be applicable to arbitrary propositional logic programs[4]. Due to this, given any propositional logic formula F, we will often refer to F as an (extended) answer set program and the propositional models of $SM[F]$ as answer sets.

[4] In fact, as shown by Cabalar and Ferraris (2007), it is possible to reduce arbitrary propositional formulas under the Stable Model Semantics to disjunctive Answer Set Programs.

3.3 A Note on Multi-Valued Formulas and the SM Semantics

Multi-valued (MV) formulas (Bartholomew and Lee (2012)) extend the traditional notion of propositional formulas with a syntax convenient for representing functions and can also be used under the Stable Model Semantics. Here we provide a brief review of the syntax of multi-valued propositional formulas and their meaning under the SM Semantics.

An *MV signature* σ is a finite set of constants such that each constant c has an associated finite domain $Dom(c)$ of at least two distinct symbols where $\sigma \cap Dom(c) = \emptyset$.

An *MV formula* is defined similar to a propositional formula except that it contains *multi-valued atoms* of the form $c = v$, such that c is a constant in σ and v is in the domain of c, in place of propositional atoms.

An *MV interpretation* A of the MV signature σ is a set of multi-valued atoms which contains exactly one such atom for each constant in σ. Satisfaction of MV formulas is defined identically to that of propositional formulas.

Given a constant $c \in \sigma$ we define the uniqueness and existence constraints for c (denoted $UEC(c)$) to be the propositional formula

$$(\neg(\bigvee_{v \in Dom(c)} c = v) \rightarrow \bot) \wedge (\bigwedge_{v_1, v_2 \in Dom(c) \text{ such that } v_1 \neq v_2} (c = v_1 \wedge c = v_2 \rightarrow \bot)).$$

Given a multi-valued signature σ we define $UEC(\sigma)$ to be $\bigwedge_{c \in \sigma} UEC(c)$.

Intuitively, the uniqueness and existence constraints for a given signature simply state that each constant within the signature must have exactly one value assigned to it.

Definition 1 (Multi-Valued ASP Programs in Propositional ASP) *Given some MV signature σ, MV formula F of σ, and MV interpretation A, A is an answer set of F if it is a model of* $\mathrm{SM}[F \wedge UEC(\sigma)]$.

Example 5 *As an example, consider the multi-valued signature* $\sigma = \{p, q\}$ *such that* $Dom(p) = \{1, 2\}$ *and* $Dom(q) = \{1, 2, 3\}$ *and multi-valued formula*

$$p = 1 \wedge (p = 1 \wedge \neg q = 1 \rightarrow (q = 2 \vee \neg q = 2)).$$

$UEC(\sigma)$ *is the conjunction of formulas*

$$\begin{aligned} &\neg(p = 1 \vee p = 2) \rightarrow \bot \\ &p = 1 \wedge p = 2 \rightarrow \bot \\ &\neg(q = 1 \vee q = 2 \vee q = 3) \rightarrow \bot \\ &q = 1 \wedge q = 2 \rightarrow \bot \\ &q = 2 \wedge q = 3 \rightarrow \bot \\ &q = 1 \wedge q = 3 \rightarrow \bot \end{aligned} \tag{3.2}$$

It holds that the only answer set of $F \wedge UEC(\sigma)$ *is then* $\{p = 1, q = 2\}$. $\{p = 1\}$ *is not an answer set as it does not satisfy* (3.2).

3.4 Action Languages

Action languages are formal models of natural language specializing in describing the effects of actions on a dynamic system (Gelfond and Lifschitz (1998)). This is done by providing a high level notation for specifying a state transition system representing the potential states of the system, the actions which may be executed, and the changes imposed by executing these actions.

Action languages differentiate themselves from formalisms such as ASP and the SM Semantics by focusing on providing enhanced usability via an intuitive, high-level language at the cost of expressivity. In practice, this is achieved by allowing the user to provide their program in the form of structured axioms that are closer to natural language than the logical statements seen in other formalisms. This, in turn, allows for use of the language without formal knowledge of the underlying semantics. In

addition, the more structured language allows for additional developer support in the form of static semantic correctness checking, such as what is seen in the system CCalc 2, which, to date, has not been adopted in ASP solving systems.

Among the existing action languages are $\mathcal{B}$ (Gelfond and Lifschitz (1998)), which allows for complex interactions between fluents within a single state; $\mathcal{C}$ (Giunchiglia and Lifschitz (1998)) which allows for richer representation of transitions between states; $\mathcal{C}+$ which extends $\mathcal{C}$ with multi-valued constants and action-dynamic laws allowing for complex interaction between simultaneous actions; and $\mathcal{BC}$ (Lee *et al.* (2013)), which integrates languages in order to allow for complex state definitions and indirect action effects.

Unfortunately, the trade off in expressivity is often stifling and prevents many useful problems from being adequately represented within each language. We consider four separate examples in order to illustrate this:

- The light switch problem described previously, which acts as a simple control example.
- A many linked switch problem in which there are a number of switches with several of them linked together. If switch 3 is on, switches 1 and 2 must assume opposite states (when one is `on`, the other must be `off`, and vice-versa). This problem evaluates the language's ability to represent recursive definitions.
- A simple pendulum problem (Giunchiglia *et al.* (2004)) in which a single pendulum moves back in forth between positions unless it is held by an agent, which demonstrates the language's ability to represent non-inertial fluents.
- The paper publishing problem described by Giunchiglia *et al.* (2004) in which an author may publish a paper with various attributes, including the length of the paper and the type of the paper (journal, conference, workshop) in an

elaboration tolerant fashion. This domain demonstrates a language's ability to handle complex relationships between actions.

None of the four languages previously mentioned are capable of adequately expressing all four domains. While $\mathcal{B}$ is able to represent recursive definitions, such as the one required in the many switches problem, it fails at representing the non-inertial state attributes and rich concurrent actions required for the pendulum and publishing problems, respectively. Conversely, $\mathcal{C}$ and $\mathcal{C}+$ are able to represent the pendulum and publishing problems, but cannot represent recursive definitions and so are unable to handle the many switches problem. Finally, $\mathcal{BC}$ is able to handle the many switches and pendulum problems, but, despite its ability to represent concurrent actions, is unable to describe the relationships between these actions required for the publishing problem.

In this section, we provide definitions for each of these languages as well as discuss their ability, or inability, to represent each of the examples previously mentioned.

In each case, we assume the presence of an MV signature consisting of disjoint sets of *fluent* and *action* symbols, denoted σ_{F} and σ_{A}, respectively. A fluent symbol can be viewed as a state attribute within a transition system, whereas an action symbol can be viewed as label for the transitions.

3.4.1 The Action Language $\mathcal{B}$

Given a multi-valued signature as previously described, we assume that each action within the signature has a domain of the Boolean values $\{\mathtt{t}, \mathtt{f}\}$.

The action language $\mathcal{B}$ was described by Gelfond and Lifschitz (1998) as having two types of laws: *static laws* of the form

$$F \textbf{ if } G \tag{3.3}$$

where F is a *fluent atom* (a multi-valued atom $c = v$ such that c is a fluent) and G is a conjunction of fluent atoms; and *dynamic laws* of the form

$$a \textbf{ causes } F \textbf{ if } H \tag{3.4}$$

where a is an action symbol, F is a fluent atom, and H is a conjunction of fluent atoms.

Intuitively, a static law (3.3) states that if G holds in the current state, then F must also hold in the current state. Similarly a dynamic law (3.4) states that if H holds in the current state and `action` is executed, then F must hold in the next state.

In all cases, we call F the *head* of the law and G and H (where applicable) the *body*. Additionally, in the event that G is $\top$ we may drop the appropriate clause for that law. As an example, the law

$$a1 \textbf{ causes } p = \texttt{t} \textbf{ if } \top$$

may be equivalently represented as

$$a1 \textbf{ causes } p = \texttt{t}.$$

We adopt a similar notation for each of the other action languages.

A $\mathcal{B}$ action description $\mathcal{D}$ is a set of static laws (3.3) and dynamic laws (3.4).

Given a formula F and time stamp t, by $t\!:\!F$ we denote the formula resulting from inserting t:in front of each fluent or action symbol occurring in F. This notation is similarly extended to sets of formulas and sets of symbols. As an example, $i\!:\!(p = 1 \wedge q = 1)$ is $i\!:\!p = 1 \wedge i\!:\!q = 1$

Given a multi-valued atom $c = v$, multi-valued constant c, or set of multi-valued constants σ, we define $\mathit{Choice}(c = v)$, $\mathit{Choice}(c)$, and $\mathit{Choice}(\sigma)$ to be

$$c = v \vee \neg c = v, \qquad \bigwedge_{v \in Dom(c)} \mathit{Choice}(c = v), \text{ and } \qquad \bigwedge_{c \in \sigma} \mathit{Choice}(c),$$

respectively.

The semantics of $\mathcal{B}$ was originally defined in terms of finding the minimal consequences of executing each action which satisfy the set of static laws. Alternatively, this process can be captured by an MV formula under the SM semantics as follows:

Definition 2 ($\mathcal{B}$ in Answer Set Programming) *Given a $\mathcal{B}$ action description $\mathcal{D}$ and some $k \geq 0$, we define the corresponding propositional formula D_k to be the conjunction of rules:*

$0{:}Choice(f)$	*for each fluent* f		
$(i-1){:}f = v \rightarrow i{:}Choice(f = v)$	*for each fluent atom* $f = v$	$(0 \leq i \leq k)$	(3.5)
$i{:}G \rightarrow i{:}F$	*for each static law*	$(0 \leq i \leq k)$	
$(i-1){:}Choice(a)$	*for each action* a	$(1 \leq i \leq k)$	(3.6)
$(i-1){:}a = \mathtt{t} \wedge (i-1){:}a' = \mathtt{t} \rightarrow \bot$	*for actions* a, a', $a \neq a'$	$(1 \leq i \leq k)$	(3.7)
$\bigwedge_{action\ a} (i-1){:}a = \mathtt{f} \rightarrow \bot$		$(1 \leq i \leq k)$	(3.8)
$(i-1){:}a = \mathtt{t} \wedge (i-1){:}H \rightarrow i{:}F$	*for each dynamic law*	$(1 \leq i \leq k)$	

Notice the inclusion of rules (3.5) and (3.6), which enforce $\mathcal{B}$'s assumption that all fluents are *inertial*, i.e. they stay the same unless caused to change, and actions are *exogenous*, allowing any action to be executed each step, respectively. In addition, the constraints (3.7) and (3.8) ensure that exactly one action is executed each step.

The transition system $\mathcal{T}(\mathcal{D})$ generated by $\mathcal{D}$ is the transition system $\mathcal{T}(\mathcal{D})$ such that

- An MV interpretation $\mathcal{S}$ of σ_{F} is a state of $\mathcal{T}(\mathcal{D})$ if $0{:}\mathcal{S}$ is an answer set of D_0, and
- Given MV interpretations $\mathcal{S}$, $\mathcal{S}'$, and $\mathcal{A}$ of signatures σ_{F}, σ_{F}, and σ_{A}, respectively, $\langle \mathcal{S}, \mathcal{A}, \mathcal{S}' \rangle$ is a transition of $\mathcal{T}(\mathcal{D})$ from $\mathcal{S}$ to $\mathcal{S}'$ labeled with $\mathcal{A}$ if $0{:}\mathcal{S} \cup 0{:}\mathcal{A} \cup 1{:}\mathcal{S}'$ is an answer set of $\mathcal{D}_1$.

Named Sets:	Value:	
Location	{`left`, `right`}	
Boolean	{`t`, `f`}	
Constants:	**Type:**	**Domain:**
Arm	Fluent	*Location*
Wait	Action	*Boolean*
Hold	Action	*Boolean*
Wait **causes** *Arm* = `left` **if** *Arm* = `right`.		$\mathcal{D}^{\mathcal{B}}_{\text{pendulum},1}$
Wait **causes** *Arm* = `right` **if** *Arm* = `left`.		$\mathcal{D}^{\mathcal{B}}_{\text{pendulum},2}$

Figure 3.1: The Pendulum Problem in $\mathcal{B}$

Of the four examples that we discussed previously, $\mathcal{B}$ is able to handle the light switch problem (Figure A.1), and the many switch problem (Figure A.2).

As mentioned previously, $\mathcal{B}$ enforces a built-in inertial assumption for each fluent in the signature. Due to this, each change in the state of the system has to be an direct or indirect effect of an action being executed. This prevents us from directly representing the swinging pendulum problem within $\mathcal{B}$. Instead, we must use a dummy *Wait* action which provides cause for the change of pendulum as is shown in Figure 3.1. However, we find this approach inadequate as it is not representative of the physical system and is not elaboration tolerant. In the event we wish to add additional actions which do not affect the movement of the pendulum, we would have to duplicate laws $\mathcal{D}^{\mathcal{B}}_{\text{pendulum},1}$ and $\mathcal{D}^{\mathcal{B}}_{\text{pendulum},2}$ for each such action.

Finally, $\mathcal{B}$ is unable to represent concurrent action execution. Due to this, all actions must be represented in a monolithic fashion as is shown in Figure 3.2. Unfortunately, this is not a desirable solution as if, for example, we wished to add a location attribute to each published paper, we would have to alter our actions to include a third tuple element as well as update each law it was referenced in.

Named Sets:	Value:		
Len	$\{1, 2, \dots\}$		
Type	$\{\texttt{journal}, \texttt{conference}, \texttt{workshop}\}$		
Boolean	$\{\texttt{t}, \texttt{f}\}$		
Constants:	**Type:**		**Domain:**
HasPub, *HasLongPub*	Fluent		*Boolean*
HasJournalPub	Fluent		*Boolean*
$Pub(l, t)$, *Wait*	Action	$(l, t) \in Len \times Type$	*Boolean*
$Pub(l, t)$ **causes** $HasPub = \texttt{t}$.		$(l, t) \in Len \times Type$	$\mathcal{D}^{\mathcal{B}}_{\text{publish},1}$
$Pub(l, t)$ **causes** $HasLongPub = \texttt{t}$.		$(l, t) \in Len \times Type$ $l > 30$	$\mathcal{D}^{\mathcal{B}}_{\text{publish},2}$
$Pub(l, \texttt{journal})$ **causes** $HasJournalPub = \texttt{t}$.		$l \in Len$	$\mathcal{D}^{\mathcal{B}}_{\text{publish},3}$
$HasPub = \texttt{t}$ **if** $HasJournalPub = \texttt{t}$.			$\mathcal{D}^{\mathcal{B}}_{\text{publish},4}$
$HasPub = \texttt{t}$ **if** $HasLongPub = \texttt{t}$.			$\mathcal{D}^{\mathcal{B}}_{\text{publish},4}$

Figure 3.2: The Publishing Problem in $\mathcal{B}$

3.4.2 The Action Language $\mathcal{C}$

The action language $\mathcal{C}$ was original described by Giunchiglia and Lifschitz (1998). As observed by Gelfond and Lifschitz, it is more general than $\mathcal{B}$ in many ways, including allowing for the representation of non-inertial fluents and concurrent action execution. It is also capable of representing concepts more succinctly by allowing for arbitrary propositional formulas in rules. However, it is not a strict generalization of $\mathcal{B}$ due to its inability to represent recursive definitions.

Giunchiglia and Lifschitz originally described $\mathcal{C}$ in terms of *causal explanation* of transitions within the transition system. More recently, a reduction has been shown from the definite fragment of $\mathcal{C}$ to propositional formulas under the SM semantics (Lifschitz and Yang (2010)). In this section, we review a multi-valued extension of the syntax of this fragment of $\mathcal{C}$ along with its ASP based semantics.

In $\mathcal{C}$, a static law is an expression of the form

$$\textbf{caused } F \textbf{ if } G \tag{3.9}$$

such that F is a fluent atom or $\bot$ and G is a *fluent formula*, an MV formula which contains no action atoms. A $\mathcal{C}$ dynamic law is an expression of the form

$$\textbf{caused } F \textbf{ if } G \textbf{ after } H \tag{3.10}$$

where F is a fluent atom or $\bot$, G is a fluent formula, and H is an MV formula.

Similar to in $\mathcal{B}$, a $\mathcal{C}$ action description $\mathcal{D}$ is a set of static and action dynamic laws.

Definition 3 ($\mathcal{C}$ in Answer Set Programming) *Given a $\mathcal{C}$ action description $\mathcal{D}$ and some $k \geq 0$, we define the corresponding propositional formula D_k to be the conjunction of rules:*

$$\begin{array}{lll}
0{:}Choice(f) & \textit{for each fluent } f & \\
\neg\neg i{:}G \rightarrow i{:}F & \textit{for each static law } (3.9) & (0 \leq i \leq k) \\
(i-1){:}Choice(a) & \textit{for each action } a & (1 \leq i \leq k) \\
(i-1){:}H \wedge \neg\neg i{:}G \rightarrow i{:}F & \textit{for each dynamic law } (3.10) & (1 \leq i \leq k)
\end{array} \tag{3.11}$$

As an example, the light switch problem may be represented in $\mathcal{C}$ as is shown in Figure A.3.

The ASP reductions of $\mathcal{B}$ and $\mathcal{C}$ share many similarities, although they have several important differences that reveal the fundamental differences in the semantics of each language. The most obvious of which is that $\mathcal{C}$ has no built in assumption of inertia, like the one provided by (3.5). Instead, the semantics of $\mathcal{C}$ allow for the action description to define which, if any, of the fluents are inertial.

This is done by including laws of the form

$$\textbf{caused } f = v \textbf{ if } f = v \textbf{ after } f = v \qquad\qquad v \in Dom(f) \tag{3.12}$$

for each inertial fluent f. When reduced to ASP, this rule becomes

$$(i-1){:}f = v \wedge \neg\neg i{:}f = v \rightarrow i{:}f = v,$$

which is strongly equivalent to

$$(i-1){:}f = v\wedge \rightarrow \mathit{Choice}(i{:}f = v).$$

For convenience, we will refer to the set of laws of the form (3.12) by the shorthand "**inertial** f."

These rules exhibit this behavior primarily because of the other significant difference in the ASP reductions of $\mathcal{B}$ and $\mathcal{C}$: in $\mathcal{C}$ the static body of each law is encased in double negation. This allows for *default* reasoning, which laws in $\mathcal{B}$ are not capable of. However, the use of double negation in this way also has a drawback in that it makes recursive definitions impossible to adequately represent.

Example 6 *For example, if we wanted to represent the many switch problem previously described, we might attempt to use an action description such as the one provided in Figure 3.3.*

However, this encoding breaks down as there is a recursive relationship between the state of switch 1 and switch 2. In $\mathcal{C}$, this will result in the state of these switches being chosen arbitrarily as long as laws $\mathcal{D}^{\mathcal{C}}_{switch2,4}$ and $\mathcal{D}^{\mathcal{C}}_{switch2,5}$ are applicable (i.e. switch 3 is on).

As $\mathcal{C}$ does not have a built-in assumption of inertia, it is possible to easily represent the pendulum problem using by defining default laws for the position of the pendulum arm which cause the pendulum to move from right to left each step, an action such as holding the pendulum can then be defined to override this default in a straight forward and extensible manner. This is shown in Figure A.4.

Named Sets:	Value:		
Status	$\{\texttt{on}, \texttt{off}\}$		
Switch	$\{\texttt{s1}, \texttt{s2}, \dots\}$		
Boolean	$\{\texttt{t}, \texttt{f}\}$		
Constants:	**Type:**		**Domain:**
$Sw(x)$	Fluent	$x \in Switch$	*Status*
$Flip(x)$	Action	$x \in Switch$	*Boolean*
inertial $Sw(x)$.		$x \in Switch$	$\mathcal{D}^{\mathcal{C}}_{\text{switch2},1}$
caused $Sw(x) = \texttt{on}$ **after** $Flip(x) = \texttt{t} \wedge Sw(x) = \texttt{off}$.		$x \in Switch$	$\mathcal{D}^{\mathcal{C}}_{\text{switch2},2}$
caused $Sw(x) = \texttt{off}$ **after** $Flip(x) = \texttt{t} \wedge Sw(x) = \texttt{on}$.		$x \in Switch$	$\mathcal{D}^{\mathcal{C}}_{\text{switch2},3}$
caused $Sw(x) = \texttt{off}$ **if** $Sw(y) = \texttt{on} \wedge Sw(\texttt{s3}) = \texttt{on}$.		$x, y \in \{\texttt{s1}, \texttt{s2}\}, x \neq y$	$\mathcal{D}^{\mathcal{C}}_{\text{switch2},4}$
caused $Sw(x) = \texttt{on}$ **if** $Sw(y) = \texttt{off} \wedge Sw(\texttt{s3}) = \texttt{on}$.		$x, y \in \{\texttt{s1}, \texttt{s2}\}, x \neq y$	$\mathcal{D}^{\mathcal{C}}_{\text{switch2},5}$

Figure 3.3: Attempting to Express the Many Switches Problem in $\mathcal{C}$

Finally, it is possible to exploit $\mathcal{C}$'s ability to represent concurrent actions in order to achieve an extensible formalization of the publishing problem as is shown in Figure 3.4. This approach, similar to the one shown for $\mathcal{C}+$ by Giunchiglia *et al.* (2004), uses additional actions, *PubType* and *PubLen*, to describe attributes of the *Publish* action. When the *Publish* does not occur, these attributes are constrained to take the value of `none` by laws $\mathcal{D}^{\mathcal{C}}_{\text{publish},4}$ and $\mathcal{D}^{\mathcal{C}}_{\text{publish},5}$. However, when *Publish* does occur, these attributes may take other values which can be reasoned on individually. If we wished to extend such a description with additional attributes, such as the location the paper was published at, it would only be necessary to add similar constraints for the new attribute and any laws that may reason about it.

Named Sets:	Value:	
Len	$\{\texttt{1}, \texttt{2}, \ldots, \texttt{none}\}$	
Type	$\{\texttt{journal}, \texttt{conference}, \texttt{workshop}, \texttt{none}\}$	
Boolean	$\{\texttt{t}, \texttt{f}\}$	
Constants:	**Type:**	**Domain:**
HasPub, *HasLongPub*	Fluent	*Boolean*
HasJournalPub	Fluent	*Boolean*
Publish	Action	*Boolean*
PubType	Action	*Type*
PubLen	Action	*Len*

inertial *HasPub*.	$\mathcal{D}^{\mathcal{C}}_{\text{publish},1}$
inertial *HasLongPub*.	$\mathcal{D}^{\mathcal{C}}_{\text{publish},2}$
inertial *HasJournalPub*.	$\mathcal{D}^{\mathcal{C}}_{\text{publish},3}$
caused $\bot$ **after** $\neg(\mathit{PubType} = \texttt{none} \leftrightarrow \mathit{Publish} = \texttt{f})$.	$\mathcal{D}^{\mathcal{C}}_{\text{publish},4}$
caused $\bot$ **after** $\neg(\mathit{PubLen} = \texttt{none} \leftrightarrow \mathit{Publish} = \texttt{f})$.	$\mathcal{D}^{\mathcal{C}}_{\text{publish},5}$
caused $\mathit{HasPub} = \texttt{t}$ **after** $\mathit{Publish} = \texttt{t}$.	$\mathcal{D}^{\mathcal{C}}_{\text{publish},6}$
caused $\mathit{HasLongPub} = \texttt{t}$ **after** $\mathit{PubLen} = x$. $\quad x \in \mathit{Len}, x > 30$	$\mathcal{D}^{\mathcal{C}}_{\text{publish},7}$
caused $\mathit{HasJournalPub} = \texttt{t}$ **after** $\mathit{PubType} = \texttt{journal}$.	$\mathcal{D}^{\mathcal{C}}_{\text{publish},8}$
caused $\bot$ **if** $\mathit{HasPub} = \texttt{f} \wedge \mathit{HasLongPub} = \texttt{t}$.	$\mathcal{D}^{\mathcal{C}}_{\text{publish},9}$
caused $\bot$ **if** $\mathit{HasPub} = \texttt{f} \wedge \mathit{HasJournalPub} = \texttt{t}$.	$\mathcal{D}^{\mathcal{C}}_{\text{publish},10}$

Figure 3.4: The Publishing Problem in $\mathcal{C}$

3.4.3 The Action Language $\mathcal{C}+$

The action language $\mathcal{C}+$ extends $\mathcal{C}$ with the ability to represent complex non-monotonic relationships between actions by allowing for non-exogenous actions and providing a new type of law in order to represent these relationships. $\mathcal{C}+$ was originally introduced by Giunchiglia *et al.* (2004) who described its semantics in terms of multi-valued non-monotonic causal theories. Later, a reduction from definite non-monotonic causal theories to first-order formulas under the SM semantics (Lifschitz and Yang (2010)) allowed for the semantics of the definite fragment $\mathcal{C}+$ to be restated directly in

terms of ASP (Lee (2012)) and gave rise to CPLUS2ASP (Casolary and Lee (2011)), which provides an efficient implementation of $\mathcal{C}+$ using this translation.

In this section we review the definite fragment of $\mathcal{C}+$ which has been implemented in the systems CCALC 2 and CPLUS2ASP.

Given a multi-valued signature divided into sets of fluent and action symbols, we further divide the fluents into disjoint sets sets of symbols $\sigma_{\text{SF}}, \sigma_{\text{SD}}$ of *simple fluents* and *statically-determined (SD) fluents*, respectively.

Intuitively, a simple fluent is similar to the fluents of $\mathcal{B}$ and $\mathcal{C}$, whereas a statically-determined fluent is one which is fully characterized by the current state of other fluents.

A $\mathcal{C}+$ *static law* is an expression of the form (3.9) such that F is a fluent atom or $\bot$ and G is a fluent formula. Similarly, an *action dynamic law* is an expression (3.9) such that F is an action atom and G is an MV formula.

A $\mathcal{C}+$ *fluent dynamic law* is an expression of the form (3.10) such that F a fluent atom $c = v$ such that c is a simple fluent or $\bot$, G is a fluent formula, and H is an MV formula.

In addition, Giunchiglia *et al.* (2004) define a number of shorthand laws, several of which are enumerated below:[5]

- "G **causes** F **if** H" stands for "**caused** F **after** $G \wedge H$" where G is an action formula;
- "**constraint** G **after** H" stands for "**caused** $\bot$ **if** $\neg G$ **after** H";
- "**nonexecutable** G **if** H" stands for "**caused** $\bot$ **after** $G \wedge H$" where G is an action formula;
- "**default** $f = v$ **if** G **after** H" is "**caused** $f = v$ **if** $f = v \wedge G$ **after** H" given a fluent atom $f = v$;

- similarly, "**default** $a = v$ **if** G" stands for "**caused** $a = v$ **if** $a = v \wedge G$" given an action atom $a = v$;
- "**inertial** f **if** G" stands for "**default** $f = v$ **if** G **after** $f = v$" for each $v \in Dom(f)$ where f is a fluent; and,
- "**exogenous** c **if** G" stands for "**default** $c = v$ **if** G" for each $v \in Dom(c)$ where c is a constant.

A $\mathcal{C}+$ action description $\mathcal{D}$ is a set of $\mathcal{C}+$ static, action dynamic, and fluent dynamic laws.

Definition 4 (Definite $\mathcal{C}+$ in Answer Set Programming) *Given a definite $\mathcal{C}+$ action description $\mathcal{D}$ and some $k \geq 0$, we define the corresponding propositional formula D_k to be the conjunction of rules:*

$0{:}Choice(f)$	*for each simple fluent* f	
$\neg\neg i{:}G \rightarrow i{:}F$	*for each static law* (3.9)	$(0 \leq i \leq k)$
$\neg\neg(i-1){:}G \rightarrow (i-1){:}F$	*for each action dynamic law* (3.9)	$(0 \leq i \leq k)$
$(i-1){:}H \wedge \neg\neg i{:}G \rightarrow i{:}F$	*for each dynamic law* (3.10)	$(1 \leq i \leq k)$

The reduction to ASP is similar to the one for $\mathcal{C}$, except that only statically determined fluents are assumed to be exogenous at the initial step and no assumption is made as to the exogeneity of actions. This, combined with the addition of action dynamic laws, allows for additional flexibility when formalizing a problem.

$\mathcal{C}+$, like $\mathcal{C}$, is able to easily represent non-inertial state changes, as is required by the pendulum problem (Figure A.6), as well as formalize non-trivial concurrent actions as are seen in the publishing problem (Figure A.7).

Unfortunately, despite the flexibility garnered by the generalizations over $\mathcal{C}$, $\mathcal{C}+$ still suffers from many of the same drawbacks as its predecessor $\mathcal{C}$. For example, $\mathcal{C}+$

[5] We refer to the reader to Appendix B of (Giunchiglia *et al.* (2004)) for the complete list of abbreviations.

runs into the same difficulties as $\mathcal{C}$ when attempting to formalize the many switches domain.

3.4.4 The Action Language $\mathcal{BC}$

Lee *et al.* (2013) define the action language $\mathcal{BC}$ to be an extension of languages $\mathcal{B}$ meant to encapsulate many of the features of $\mathcal{C}$, such as representing non-inertial fluents, while allowing for complex interactions between fluents within the same state.

As in Section 3.4.3, we assume the presence of a signature σ partitioned into sets of simple fluents, statically-determined fluents, and actions. Furthermore, we assume that all action symbols in σ have a domain of $\{\mathtt{t}, \mathtt{f}\}$.

In $\mathcal{BC}$, a *static law* is an expression of the form

$$F \textbf{ if } G_1 \textbf{ ifcons } G_2 \tag{3.13}$$

where F is a fluent atom, and G_1 and G_2 are conjunctions of fluent atoms. A $\mathcal{BC}$ *dynamic law* is an expression of the form

$$F \textbf{ if } G_1 \textbf{ ifcons } G_2 \textbf{ after } H \tag{3.14}$$

where F is a fluent atom, G_1 and G_2 are conjunctions of fluent atoms, and H is a conjunction of fluent atoms and action symbols.

Intuitively, (3.13) is read "If G_1 is satisfied in the current state and it is consistent to assume that G_2 is satisfied, then F must also be satisfied in the current state". Similarly, (3.14) states "If the fluent atoms in H were satisfied in the last state and the action symbols occurred in the transition, G_1 is satisfied in the current state, and it is consistent to assume that G_2 is satisfied in the current state, then F must also be satisfied in the current state.

In addition, Lee *et al.* provide a number of shorthand laws as follows:

- "a **causes** F **if** H" stands for "F **after** $a \wedge H$" where a is an action symbol;
- "**impossible** G" stands for the laws

$$f = v \textbf{ if } G, \text{ and}$$
$$f = w \textbf{ if } G$$

  for some fluent f and $v, w \in Dom(f)$ such that $v \neq w$;
- given a conjunction of action symbols G, "**nonexecutable** G **if** H" stands for

$$f = v \textbf{ after } G \wedge H, \text{ and}$$
$$f = w \textbf{ after } G \wedge H$$

  for some fluent f and $v, w \in Dom(f)$ such that $v \neq w$;
- "**default** $f = v$ **if** G **after** H" stands for "$f = v$ **if** G **ifcons** $f = v$ **after** H"; and
- "**inertial** f" stands for "**default** $f = v$ **after** $f = v$" for each $v \in Dom(f)$ where f is a fluent.

A $\mathcal{BC}$ action description $\mathcal{D}$ is a set of $\mathcal{BC}$ static and fluent dynamic laws.

Lee *et al.* defined $\mathcal{BC}$ in terms of a reduction to formulas under the stable model semantics as follows:

Definition 5 ($\mathcal{BC}$ in Answer Set Programming) *Given a $\mathcal{BC}$ action description $\mathcal{D}$ and some $k \geq 0$, we define the corresponding propositional formula D_k to be the conjunction of rules:*

$0{:}Choice(f)$	*for each simple fluent* f	
$i{:}G_1 \wedge \neg\neg i{:}G_2 \rightarrow i{:}F$	*for each static law* (3.13)	$(0 \leq i \leq k)$
$(i-1){:}Choice(a)$	*for each action* a	$(1 \leq i \leq k)$
$(i-1){:}H \wedge i{:}G_1 \wedge \neg\neg i{:}G_2 \rightarrow i{:}F$	*for each dynamic law* (3.14)	$(1 \leq i \leq k)$

Named Sets:	Value:	
Location	{`left`, `right`}	
Boolean	{`t`, `f`}	
Constants:	**Type:**	**Domain:**
Arm	Fluent	*Location*
Hold	Action	*Boolean*
default $Arm = \texttt{left}$ **after** $Arm = \texttt{right}$		$\mathcal{D}^{\mathcal{BC}}_{\text{pendulum},1}$
default $Arm = \texttt{right}$ **after** $Arm = \texttt{left}$		$\mathcal{D}^{\mathcal{BC}}_{\text{pendulum},2}$
Hold **causes** $Arm = \texttt{right}$ **if** $Arm = \texttt{right}$.		$\mathcal{D}^{\mathcal{BC}}_{\text{pendulum},3}$
Hold **causes** $Arm = \texttt{left}$ **if** $Arm = \texttt{left}$.		$\mathcal{D}^{\mathcal{BC}}_{\text{pendulum},4}$

Figure 3.5: The Pendulum Problem in $\mathcal{BC}$

where each action symbol a is understood as shorthand for the atom $a = \texttt{t}$.

The translation is similar to that of $\mathcal{B}$, with several key differences:

- no equivalent to (3.5) is generated, allowing for non-inertial fluents;
- similarly, no equivalent to (3.7) is generated, allowing for concurrent action execution; and
- each law has a new "ifcons" clause which is placed in the scope of double negation, similar to the "if" clauses within $\mathcal{C}$ and $\mathcal{C}+$.

As it turns out, the $\mathcal{C}$ style "ifcons" clause within $\mathcal{BC}$ is crucial to the representation of non-inertial fluents, as it provides a way to represent the concept of inertia within an action description, which the strong assumption that all fluents are inertial to be dropped from the semantics of the language.

$\mathcal{BC}$, like $\mathcal{B}$, is able to easily represent the recursive relationships showcased in the many switches problem. In fact, the encodings of the light switch and many switches problems (Figure A.8 and A.9) are unremarkably similar to their $\mathcal{B}$ counterparts, with the only difference being the explicit representation of inertia.

Additionally, unlike $\mathcal{B}$, $\mathcal{BC}$ is able to represent non-inertial changes to the state using default rules as was showcased by $\mathcal{C}$ and $\mathcal{C}+$. This allows $\mathcal{BC}$ to easily represent problems such as the swinging pendulum problem, as is shown in Figure 3.5.

Although $\mathcal{BC}$ is able to represent concurrent actions, its ability to represent complex relationships between these actions is extremely limited. In fact, the only relationships that can be represented is combinations of actions that should not be executed in each state. Unfortunately, this is not expressive enough for elaboration tolerant action attributes, as are showcased in the publishing problem. Due to this, $\mathcal{BC}$ must approach the publishing problem in the same was as $\mathcal{B}$ and use a single monolithic action for each combination of attributes.

Chapter 4

THE ACTION LANGUAGE $\mathcal{BC}+$: INTEGRATING ASP, $\mathcal{C}+$, AND $\mathcal{BC}$

As we observed previously, of the four action languages we have reviewed, none have been able to adequately represent recursive relationships, non-inertial/default state changes, and rich concurrent action relationships. However, each of these concepts can be represented in ASP.

On the other hand, although ASP provides an extremely expressive formalism it lacks much of the structure and support present in modern programming languages and integrated development environments. Instead, ASP is more of a "logical assembly language", capable of providing a unified backbone for execution, yet largely unsuitable for crafting problem descriptions of significant size.

In order to attempt to remedy these deficiencies, we propose a new action language, $\mathcal{BC}+$, which provides a proper generalization for definite $\mathcal{C}+$ and $\mathcal{BC}$ by leveraging the full expressivity of modern ASP, all the while providing a traditional action language structure in order to enhance the accessibility of the formalism in terms of readability and supportability.

4.1 Defining the Action Language $\mathcal{BC}+$

We assume the presence of a signature σ divided into distinct sets of simple fluents, statically determined fluents, and actions, as in Section 3.4.3.

A $\mathcal{BC}+$ *static law* is a law of the form

$$F \textbf{ if } G \tag{4.1}$$

such that F is a fluent atom, $\bot$, or fluent formula of the form $\mathit{Choice}(c = v)$, and G is a fluent formula (MV formula). Similarly, a $\mathcal{BC}+$ *action dynamic law* is a law

of the form (4.1) such that F is an action atom, $\bot$, or action formula of the form $\mathit{Choice}(c = v)$, and G is an MV formula.

A $\mathcal{BC}+$ *fluent dynamic law* is a law of the form

$$F \textbf{ if } G \textbf{ after } H \tag{4.2}$$

such that F is a fluent atom $c = v$ or fluent formula of the form $\mathit{Choice}(c = v)$ where c is a simple fluent or $\bot$, G is a fluent formula, and H is an MV formula.

Similar to in $\mathcal{C}+$, we define the following shorthand laws:

$a = v$ **causes** F **if** H	$\mapsto$	F **after** $a = v \wedge\ H$	
impossible G **after** H	$\mapsto$	$\bot$ **if** G **after** H	
nonexecutable $a = v$ **if** H	$\mapsto$	$\bot$ **if** $a = v \wedge H$	
default $c = v$ **if** G **after** H	$\mapsto$	$\mathit{Choice}(c = v)$ **if** G **after** H	
inertial f **if** G	$\mapsto$	**default** $f = v$ **if** G **after** $f = v$	$(v \in \mathit{Dom}(f))$
exogenous c **if** G **after** H	$\mapsto$	**default** $c = v$ **if** G **after** H	$(v \in \mathit{Dom}(c))$

where f is a fluent constant, c is a constant, and a is an action constant.

A $\mathcal{BC}+$ *action description* $\mathcal{D} = \mathcal{D}_{\mathrm{S}} \cup \mathcal{D}_{\mathrm{AD}} \cup \mathcal{D}_{\mathrm{FD}}$ consists of a finite set of static laws $\mathcal{D}_S$, action dynamic laws $\mathcal{D}_{\mathrm{AD}}$, and fluent dynamic laws $\mathcal{D}_{\mathrm{FD}}$.

As an example, the toggle switch problem previously discussed may be represented in $\mathcal{BC}+$ as is shown in Figure 4.1.

Named Sets:	Value:		
Status	{`on`, `off`}		
Boolean	{`t`, `f`}		
Constants:	**Type:**		**Domain:**
Sw	Simple Fluent		*Status*
Light	SD Fluent		*Status*
Flip	Action		*Boolean*
inertial *Sw*.			$\mathcal{D}^{\mathcal{BC}+}_{\text{switch},1}$
exogenous *Flip*.			$\mathcal{D}^{\mathcal{BC}+}_{\text{switch},2}$
Flip = `t` **causes** *Sw* = `on` **if** *Sw* = `off`.			$\mathcal{D}^{\mathcal{BC}+}_{\text{switch},3}$
Flip = `t` **causes** *Sw* = `off` **if** *Sw* = `on`.			$\mathcal{D}^{\mathcal{BC}+}_{\text{switch},4}$
default *Light* = *s* **if** *Sw* = *s*.		$s \in$ *Status*	$\mathcal{D}^{\mathcal{BC}+}_{\text{switch},5}$

Figure 4.1: Toggle Switch Domain in $\mathcal{BC}+$.

4.2 A Transition System Based Semantics

Given a $\mathcal{BC}+$ action description $\mathcal{D}$, we divide the process of transition system construction into three parts: we first determine the states of $\mathcal{D}$, afterward, for each state, we find the set of potential transition labels leaving the state, finally, we characterize the final state (or states), if one exists, resulting from following each label from its originating state.

Given a problem description $\mathcal{D}$, we say an MV interpretation $\mathcal{S}$ of σ_{F} is a *state* of $\mathcal{D}$ if $\mathcal{S}$ is an answer set of

$$\bigwedge_{(4.1)\in\mathcal{D}_{\text{S}}} (G \rightarrow F) \wedge \mathit{Choice}(\sigma_{\text{SF}}). \tag{4.3}$$

Given a state $\mathcal{S}$, we say an MV interpretation $\mathcal{A}$ of σ_{A} is a *candidate transition label* leaving $\mathcal{S}$ if $\mathcal{S} \cup \mathcal{A}$ is an answer set of

$$\bigwedge_{(4.1)\in\mathcal{D}_{\text{AD}}} (G \rightarrow F) \wedge \mathit{Choice}(\sigma_{\text{F}}). \tag{4.4}$$

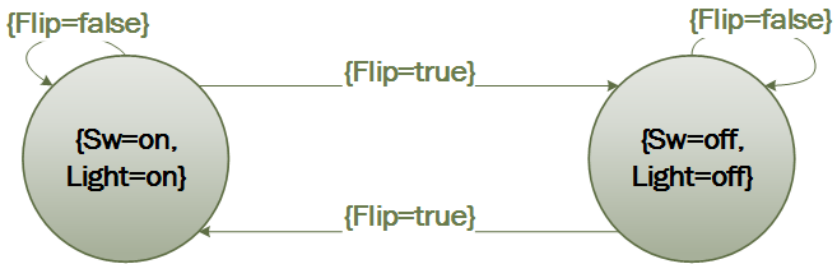

Figure 4.2: The transition system of $\mathcal{D}_{\text{switch}}$

Given states $\mathcal{S}$ and $\mathcal{S}'$ and a candidate transition label $\mathcal{A}$ leaving $\mathcal{S}$, we say $\mathcal{T} = \langle \mathcal{S}, \mathcal{A}, \mathcal{S}' \rangle$ is a *transition* of $\mathcal{D}$ if $0{:}\mathcal{S} \cup 0{:}\mathcal{A} \cup 1{:}\mathcal{S}'$ is an answer set of

$$\bigwedge_{(4.2)\in\mathcal{D}_{\text{FD}}} 0{:}M \wedge 1{:}G \rightarrow 1{:}F \wedge \bigwedge_{(4.1)\in\mathcal{D}_{\text{S}}} (1{:}G \rightarrow 1{:}F) \wedge \mathit{Choice}(0{:}\sigma_{\text{F}} \cup 0{:}\sigma_{\text{A}}). \tag{4.5}$$

Example 7 *Consider the $\mathcal{D}_{switch}$ problem description provided in Figure 4.1. The only static law present is $\mathcal{D}_{switch,5}$, the only action dynamic law is $\mathcal{D}_{switch,2}$, while the remaining laws are fluent dynamic.*

The states of the transition system are then answer sets of signature $\{Sw, Light\}$ of

$$(Sw = \texttt{off} \rightarrow Light = \texttt{off}) \wedge \mathit{Choice}(Sw).$$

There are two such interpretations:

$$\mathcal{S}_0 = \{Sw = \texttt{off}, Light = \texttt{off}\}, \text{ and} \qquad \mathcal{S}_1 = \{Sw = \texttt{on}, Light = \texttt{on}\}.$$

As $\mathcal{D}_{switch,2}$ is the only action dynamic law and contains no fluent constants, it is clear that all states have the same set of candidate transition labels which are the answer sets of

$$\bigwedge_{b\in\{\texttt{true},\texttt{false}\}} \mathit{Choice}(Flip = b).$$

This means that there are two possible transition labels leaving each state:

$$\mathcal{A}_0 = \{Flip = \texttt{true}\}, \text{ and} \qquad \mathcal{A}_1 = \{Flip = \texttt{false}\}.$$

Finally, the transitions of $\mathcal{D}$ are tuples $\langle \mathcal{S}, \mathcal{A}, \mathcal{S}' \rangle$ of states and candidate action labels such that $0{:}\mathcal{S} \cup 0{:}\mathcal{A} \cup 1{:}\mathcal{S}'$ is an answer set of

$$\bigwedge_{s \in \{\texttt{on},\texttt{off}\}} (0{:}Sw = s \rightarrow Choice(1{:}Sw = s)) \qquad \mathcal{D}_{switch,1}$$

$$\wedge\, (0{:}Flip = \texttt{true} \wedge 0{:}Sw = \texttt{on} \rightarrow 1{:}Sw = \texttt{off}) \qquad \mathcal{D}_{switch,3}$$

$$\wedge\, (0{:}Flip = \texttt{true} \wedge 0{:}Sw = \texttt{off} \rightarrow 1{:}Sw = \texttt{on}) \qquad \mathcal{D}_{switch,4}$$

$$\wedge\, ((1{:}Sw = \texttt{on} \rightarrow 1{:}Choice(Light = \texttt{on}))) \qquad \mathcal{D}_{switch,5}$$

$$\wedge\, Choice(\{0{:}Sw, 0{:}Light, 0{:}Flip\})$$

In this case, there is exactly one successor state $\mathcal{S}'$ for each initial state $\mathcal{S}$ and candidate transition label $\mathcal{A}$ combination, creating 4 transitions. Each of these transitions are displayed in the final transition system provided in Figure 4.2.

Definition 6 ($\mathcal{BC}+$ Histories) *A history $\mathcal{H}_k$ of $\mathcal{D}$ is a path of length k through the transition system $\mathcal{T}(\mathcal{D})$. We may also identify $\mathcal{H}_k$ with the MV interpretation*

$$0{:}\mathcal{S}_0 \cup 0{:}\mathcal{A}_0 \cup 1{:}\mathcal{S}_1 \cup 1{:}\mathcal{A}_1 \cup \cdots \cup k{:}\mathcal{S}_k$$

where each $\mathcal{S}_i$ $(0 \leq i \leq k)$ is the ith state visited and each $\mathcal{A}_j$ $(0 \leq j < k)$ is the jth transition label taken in the history.

4.3 An ASP Based Semantics

Similar to many other actions languages, a $\mathcal{BC}+$ action description may also represented in terms of its reduction into an MV formula under the Stable Model Semantics.

Definition 7 (Capturing $\mathcal{BC}+$ in ASP) *Given a $\mathcal{BC}+$ action description $\mathcal{D}$ and some $k \geq 0$, we define D_k to be the conjunction of rules*

$$
\begin{array}{lll}
0{:}Choice(f) & \textit{for each simple fluent } f & \qquad (4.6) \\
i{:}G \rightarrow i{:}F & \textit{for each static law } (4.1) & (0 \leq i \leq k) \\
(i-1){:}G \rightarrow (i-1){:}F & \textit{for each action dynamic law } (4.1) & (1 \leq i \leq k) \qquad (4.7) \\
(i-1){:}H \wedge i{:}G \rightarrow i{:}F & \textit{for each fluent dynamic law } (4.2) & (1 \leq i \leq k)
\end{array}
$$

As it turns out, this approach is equivalent to the construction of the transition system $\mathcal{T}(\mathcal{D})$ presented in the previous section. This is stated formally in Proposition 2.

Proposition 2 (Equivalence of Semantics) *Given a $\mathcal{BC}+$ action description $\mathcal{D}$ and any $k \geq 0$ it holds that the histories of length k of $\mathcal{T}(\mathcal{D})$ correspond exactly to the answer sets of D_k.*

As a special case, we can then derive a definition of our $\mathcal{BC}+$ transition system similar to those provided for each of the other action languages.

Corollary 1 *Given a $\mathcal{BC}+$ action description $\mathcal{D}$, it holds that the answer sets D_0 and D_1 correspond to the states and transition of $\mathcal{T}(\mathcal{D})$, respectively.*

Example 8 *As an example, given the $\mathcal{BC}+$ toggle switch action description $\mathcal{D}_{switch}$, $D_{switch,1}$ is*

$$\bigwedge_{s\in\{\texttt{on},\texttt{off}\}} \mathit{Choice}(0{:}\mathit{Sw} = \ s)$$

$$\wedge \bigwedge_{s\in\{\texttt{on},\texttt{off}\}} (0{:}\mathit{Sw} = \ s \rightarrow \mathit{Choice}(0{:}\mathit{Light} = \ s)) \qquad \mathcal{D}_{switch,5}$$

$$\wedge \bigwedge_{b\in\{\texttt{true},\texttt{false}\}} 0{:}\mathit{Choice}(\mathit{Flip} = b) \qquad \mathcal{D}_{switch,2}$$

$$\wedge \bigwedge_{s\in\{\texttt{on},\texttt{off}\}} (1{:}\mathit{Sw} = \ s \rightarrow \mathit{Choice}(1{:}\mathit{Light} = \ s)) \qquad \mathcal{D}_{switch,5}$$

$$\wedge \bigwedge_{s\in\{\texttt{on},\texttt{off}\}} (0{:}\mathit{Sw} = s \rightarrow \mathit{Choice}(1{:}\mathit{Sw} = s)) \qquad \mathcal{D}_{switch,1}$$

$$\wedge (0{:}\mathit{Flip} = \texttt{true} \wedge 0{:}\mathit{Sw} = \texttt{off} \rightarrow 1{:}\mathit{Sw} = \texttt{on}) \qquad \mathcal{D}_{switch,3}$$

$$\wedge (0{:}\mathit{Flip} = \texttt{true} \wedge 0{:}\mathit{Sw} = \texttt{on} \rightarrow 1{:}\mathit{Sw} = \texttt{off}) \qquad \mathcal{D}_{switch,4}$$

Which has 4 answer sets

$A_0 = \{0{:}\mathit{Sw} = \texttt{off}, 0{:}\mathit{Light} = \texttt{off}, 0{:}\mathit{Flip} = \texttt{false}, 1{:}\mathit{Sw} = \texttt{off}, 1{:}\mathit{Light} = \texttt{off}\}$,

$A_1 = \{0{:}\mathit{Sw} = \texttt{off}, 0{:}\mathit{Light} = \texttt{off}, 0{:}\mathit{Flip} = \texttt{true}, 1{:}\mathit{Sw} = \texttt{on}, 1{:}\mathit{Light} = \texttt{on}\}$,

$A_2 = \{0{:}\mathit{Sw} = \texttt{on}, 0{:}\mathit{Light} = \texttt{on}, 0{:}\mathit{Flip} = \texttt{false}, 1{:}\mathit{Sw} = \texttt{on}, 1{:}\mathit{Light} = \texttt{on}\}$, *and*

$A_3 = \{0{:}\mathit{Sw} = \texttt{on}, 0{:}\mathit{Light} = \texttt{on}, 0{:}\mathit{Flip} = \texttt{true}, 1{:}\mathit{Sw} = \texttt{off}, 1{:}\mathit{Light} = \texttt{off}\}$.

These correspond exactly to the transitions described in Example 7.

4.4 Relation to Existing Formalisms

Like $\mathcal{B}$ and $\mathcal{BC}$, $\mathcal{BC}+$ is able to represent recursive relationships, such as those exhibited in the many switch problem, a formalization of which is shown in Figure A.11. In addition, $\mathcal{BC}+$ is able to match the expressivity of $\mathcal{C}$ and $\mathcal{C}+$ in regards to representing non-inertial fluents and complex concurrent actions, which is exhibited by the implementations of the pendulum and publishing problems shown in Figures A.12 and A.13.

In fact, $\mathcal{BC}+$ is a proper generalization of each of the languages reviewed in Section 3.4. In this section, we provide the mapping from definite $\mathcal{C}+$ and $\mathcal{BC}$ to $\mathcal{BC}+$. Given these mappings, the mappings for $\mathcal{C}$ and $\mathcal{B}$ then follow immediately as special cases.

Given a definite $\mathcal{C}+$ action description $\mathcal{D}$, we define the corresponding $\mathcal{BC}+$ action description $cp2bcp(\mathcal{D})$ to be the description obtained by replacing each static and action dynamic law (3.9) with the law

$$F \textbf{ if } \neg\neg G$$

and each fluent dynamic law (3.10) with the law

$$F \textbf{ if } \neg\neg G \textbf{ after } H.$$

The intuition behind this mapping is quite clear. By examining the reduction of both $\mathcal{C}+$ and $\mathcal{BC}+$ into ASP, we can easily see that the only difference in them is the presence of the implicit double negation in front of the static bodies (G) of each of the laws. Therefore, adding this double negation makes the two resulting ASP programs equivalent. This is stated formally in Proposition 3.

Proposition 3 *Given a definite $\mathcal{C}+$ action description $\mathcal{D}$, it holds that the transition system corresponding to $\mathcal{D}$ is exactly $\mathcal{T}(cp2bcp(\mathcal{D}))$.*

Given a $\mathcal{BC}$ action description $\mathcal{D}$, we define the corresponding $\mathcal{BC}+$ action description $bc2bcp(\mathcal{D})$ to be the description obtained by understanding " **ifcons** G_2" as shorthand for appending "$\neg\neg G_2$" to the law's if clause and adding the action dynamic law "$Choice(a)$" for each action $a \in \sigma$.

Proposition 4 *Given a $\mathcal{BC}$ action description $\mathcal{D}$, it holds that the transition system corresponding to $\mathcal{D}$ is exactly $\mathcal{T}(bc2bcp(\mathcal{D}))$.*

Not only is $\mathcal{BC}+$ a generalization of each of the action languages reviewed in Section 3.4, it can also be trivially observed that it is as expressive as propositional answer set programs. This can be done by understanding each constant in the ASP program as a Boolean statically determined fluent constant, expressing each ASP rule

$$G \rightarrow F$$

as the static law

$$F \textbf{ if } G,$$

and adding the law

$$\textbf{default } c = \mathtt{f}$$

for each constant c.

The states of the resulting action description are then exactly the answer sets of the initial ASP program.

Chapter 5

MODULAR AND ONLINE ANSWER SET PROGRAMS

Traditionally, KR formalisms have largely ignored the execution phase of problem solving and instead focused exclusively on providing tools to assist planning, rather than execution, assuming that the system will behave exactly as expected. However, what if an unexpected event occurs? Or an action performed by the agent has unintended consequences? An agent operating in most real world environments has to respond to external input and events which are not known a priori.

One approach to handling this is to treat each of these events as an exception which causes the agent to create a new plan based on the current state of the system and the new information. In the past, this has been done by completely throwing out the previous results and restarting the planning process from scratch. Unfortunately, while this works in small examples, it is not feasible in large or time sensitive problems as it exasperates the inherent intractability of solving ASP programs.

To make matters worse, planning problems often involve attempting to find the minimum length plan to achieve the goal. In order to do this, an iterative deepening search is employed in which the system first checks for a plan of length 0, followed by one of length 1, and so on and so forth. Similar to handling execution exceptions, each of these iterations historically requires the grounding/solving process to be completely restarted. This means that, in the event the event the minimum plan is of length k, the ASP program is re-grounded and resolved k times *each time new information is acquired during execution.*

Recently, the systems iClingo (Gebser *et al.* (2008)) and oClingo (Gebser *et al.* (2011a)) have been introduced in order to attempt to solve these issues. iClingo

allows the user to specify a program template consisting of three components: the *base component*, which contains static information, such as the initial state of a system; the *incremental component*, which contains step variant knowledge, such as how a system will evolve over time; and the *volatile component*, which contains information about the final step, such as the goal that should be reached (i.e. desirable attributes of the final state). iClingo uses the Module Theorem (Janhunen *et al.* (2007);Oikarinen and Janhunen (2006)) to perform an iterative deepening search, such as the one required for finding the minimum length plan to achieve a goal, without requiring a restart for the grounding/solving process. Instead, the new step is grounded and composed to the existing program and solving continues using previously learned heuristic information. oClingo takes the work done by iClingo a step further by extending their theory to allow for online information to be dynamically added to the system as a substitute to restarting the process when an execution exception occurs.

In practice, these improvements result in a drastic performance increase when considering problems which require an iterative deepening approach and/or desire fault tolerant execution. Although this approach shows a lot of promise for improving the scalability of ASP systems, iClingo and oClingo assume their input programs are *modular* and *mutually revisable*, and do not verify them to ensure compliance. These conditions require intimate knowledge of the theory behind these systems and impose additional difficulties for a potential developed. The result is a drastic increase in developmental complexity which makes the systems undesirable.

In this chapter we review the background theories behind these systems in preparation for extending them in order to formalize an online extension to the proposed $\mathcal{BC}+$ language which helps to abstract the design complexity involved in creating an online ASP program.

We begin by reviewing the Splitting Theorem (Ferraris *et al.* (2009a)) and the Module Theorem. These theorems both allow for larger problems to be divided into smaller ones and considered individually. Although similar in nature, neither subsume the other in terms of generality, and, until recently, their relationship has not been formalized. Following this review, we present a first-order generalization of the Module Theorem which also fully subsumes the Splitting Theorem. Following this, we provide a review of the online ASP theory behind system oCLINGO, which generalizes iCLINGO's incremental theory.

5.1 The Symmetric Splitting Theorem

Initially, the Splitting Theorem was defined by Lifschitz and Turner (1994) as a means to consider disjunctive ASP programs consisting of rules of the form (2.3) by splitting it along asymmetric splitting sets. Later, Ferraris *et al.* (2009a) introduced a symmetric extension to this theorem to be applicable for first-order formulas under the Stable Model Semantics. This generalized version examines the *predicate dependency graph* of a program, which represents the positive relationships between each of the predicate symbols in the formula, in order to identify strongly connected components. According to Ferraris *et al.*, each of these strongly connected components may then be separated out and considered independently.

The following review follows Ferraris *et al.*'s definition.

Formally, we identify an occurrence of a subformula as *positive* if the number of implications[1] containing the occurrence in the antecedent is even. Furthermore, we say the subformula is *strictly positive* if that number is 0. Finally, we say a formula F is *negative* on a list of predicate symbols $\mathbf{p}$ if there is no strictly positive occurrence of any $p \in \mathbf{p}$ within F.

[1] Recall that we treat the formula $\neg F$ as $F \rightarrow \bot$.

As an example, consider the formula

$$p(a) \wedge q(b) \wedge \forall x((\neg s \wedge \neg q(x) \wedge p(x)) \rightarrow r(x)). \tag{5.1}$$

Both occurrences of q in (5.1) are positive, however only first is strictly positive. In addition, (5.1) is negative on $\{s\}$.

Definition 8 (Predicate Dependency Graph, Ferraris *et al.* (2009a)) *The predicate dependency graph of F relative to a list of predicate symbols $\mathbf{p}$, denoted* $\mathrm{DG}[F; \mathbf{p}]$*, is the directed graph that*

- *has all members of $\mathbf{p}$ as its vertices, and*
- *has an edge from p to q if, for some rule $G \rightarrow H$ of F,*
  - *p has a strictly positive occurrence in H, and*
  - *q has a positive occurrence in G that does not belong to any subformula of G that is negative on $\mathbf{p}$.*

As mentioned previously, the predicate dependency graph is essentially used to model the positive relationships of predicates within a formula. The intuition behind this is that, as we saw in Example 4, an occurrence of a subformula R which is negative on the intensional predicates will not be replaced by $R^*(\mathbf{u})$ while calculating $F^*(\mathbf{u})$, which serves to fix its interpretation when considering if the model is minimal. Due to this, an edge exists from p to q in the dependency graph only if the occurrence of q is not in such a subformula.

For example, $\mathrm{DG}[(5.1); pqr]$ has the vertices p, q, and r, and a single edge from r to p.

Theorem 1 (The Splitting Theorem, Ferraris *et al.* (2009a)) *Let F and G be first-order sentences, and let $\mathbf{p}$ and $\mathbf{q}$ be finite disjoint lists of distinct predicate constants. If*

(a) each strongly connected component of the predicate dependency graph of $F \wedge G$ *relative to* $\mathbf{p}$, $\mathbf{q}$ *is either a subset of* $\mathbf{p}$ *or a subset of* $\mathbf{q}$,

(b) F *is negative on* $\mathbf{q}$, *and*

(c) G *is negative on* $\mathbf{p}$

then

$$\mathrm{SM}[F \wedge G;\ \mathbf{p} \cup \mathbf{q}] \leftrightarrow \mathrm{SM}[F;\ \mathbf{p}] \wedge \mathrm{SM}[G;\ \mathbf{q}]$$

is logically valid.

Theorem 1 tells us that SM[(5.1)] is equivalent to

$$\mathrm{SM}[p(a); p] \wedge \mathrm{SM}[q(b); q] \wedge \mathrm{SM}[\forall x(\neg s \wedge p(x) \wedge \neg q(x) \rightarrow r(x)); r].$$

Which allows us to evaluate the stable models of $p(a)$, $q(b)$ and

$$\forall x(\neg s \wedge p(x) \wedge \neg q(x) \rightarrow r(x))$$

independently. In the event an interpretation is a stable model of all three, the Splitting Theorem tells us that it must be a stable model of (5.1).

5.2 The Module Theorem for Disjunctive Answer Set Programs

The Module Theorem was introduced by Oikarinen and Janhunen (2006) as an extension to Lifschitz and Turner (1994)'s Splitting Theorem. It acts as a means to allow solutions of smaller problems to be composed to create solutions to the problem at large. The Module Theorem primarily distinguishes itself from Ferraris *et al.* (2009a)'s Splitting Theorem in two ways: like the original splitting theorem, it is limited to disjunctive answer set programs; and, each submodule is described in a *module* with subsets of the overarching signature which allows for solutions of each module to be composed to generate solutions for the final program. (This

differentiates itself from the Splitting Theorem which only allows solutions to be checked against each sub-problem).

Given a formula F we define $Pred(F)$ to be the set of predicates which F. Note that in the event F is propositional $Pred(F)$ is the set propositional atoms within F.

A *DLP-module* is a triple $\langle F, I, O\rangle$ where F is a disjunctive logic program, and I and O are finite, disjoint sets of propositional atoms such that $Pred(F) \subseteq I \cup O$.

Definition 9 (Module Answer Sets, Janhunen *et al.* (2009)) *We say that a set X of atoms is a (module) answer set of a DLP-module $\langle F, I, O\rangle$ if X is an answer set of $F \wedge Choice(I)$.*

Intuitively, for any DLP-module $\langle F, I, O\rangle$, its output atoms are those that are characterized by the module, while input atoms are external in the sense that it is expected to be characterized by a separate module. Thus, when we interpret $\langle F, I, O\rangle$ we allow the input atoms to be asserted arbitrarily (as we have no additional regarding these atoms) while minimizing the occurrences of output atoms.

Given two or more of these DLP-modules, it is possible to compose them into a larger DLP-module which characterizes each of the atoms originally characterized by its progenitors. This process, known as *joining* the modules, imposes a precondition very similar to the splitting theorem by examining the predicate dependency graph for strongly connected components which span each of the modules. This is defined formally in Definition 10.

Definition 10 (Join of Modules) *Two DLP-modules $\mathbb{F}_1 = (F_1, I_1, O_1)$ and $\mathbb{F}_2 = (F_2, I_2, O_2)$ are called joinable if*

- *each strongly connected component of* $\mathrm{DG}[F_1 \wedge F_2;\ O_1 \cup O_2]$ *is either a subset of O_1 or a subset of O_2,*

- *any rule in F_1 with a head atom in O_2 also occurs in F_2, and symmetrically*
- *any rule in F_2 with any head atom in O_1 also occurs in F_1.*

Given two such DLP-modules, the join of $\mathbb{F}_1$ and $\mathbb{F}_2$, denoted by $\mathbb{F}_1 \sqcup \mathbb{F}_2$, is defined as the DLP-module

$$\langle F_1 \wedge F_2, (I_1 \cup I_2) \setminus (O_1 \cup O_2), O_1 \cup O_2 \rangle$$

The notable difference between the preconditions for the joining modules and that imposed by the Splitting Theorem is that the Module Theorem allows certain rules to overlap between modules. As an example, consider the DLP-modules

$$\langle (p \vee q) \wedge (p \rightarrow r) \wedge s, \{q\}, \{p, r, s\} \rangle, \text{ and} \tag{5.2}$$

$$\langle (p \vee q) \wedge (\neg r \vee \neg p), \{p, r\}, \{q\} \rangle. \tag{5.3}$$

(5.2) and (5.3) are joinable and result in the DLP-module

$$\langle (p \vee q) \wedge (p \rightarrow r) \wedge s \wedge (\neg r \vee \neg p), \emptyset, \{p, q, r, s\} \rangle \tag{5.4}$$

which has a single answer set $\{q, s\}$. However, $SM[(p \vee q) \wedge (p \rightarrow r); prs]$ and $SM[(p \vee q) \wedge (\neg r \vee \neg p); q]$ cannot be composed via the Splitting Theorem.

Given sets of atoms X_1, X_2, and A, we say that X_1 and X_2 are *A-compatible* if they agree on the set of atoms A, that is $X_1 \cap A = X_2 \cap A$.

Theorem 2 (The Module Theorem for Disjunctive Answer Set Programs) *Let $\mathbb{F}_1 = (F_1, I_1, O_1)$ and $\mathbb{F}_2 = (F_2, I_2, O_2)$ be DLP-modules such that $\mathbb{F}_1$ and $\mathbb{F}_2$ are joinable and let X_1 and X_2 be $((I_1 \cup O_1) \cap (I_2 \cup O_2))$-compatible sets of atoms. The set $X_1 \cup X_2$ is a module answer set of $\mathbb{F}_1 \sqcup \mathbb{F}_2$ iff X_1 is a module answer set of $\mathbb{F}_1$ and X_2 is a module answer set of $\mathbb{F}_2$.*

As an example, (5.2) has the answer sets $\{q, s\}$, $\{p, r, s\}$ and $\{p, q, r, s\}$, while (5.2) has the answer sets $\{p\}$, $\{q\}$, and $\{q, r\}$ and $\{q\}$. Between these, the only $\{p, q, r\}$ compatible pair is $\{q, s\}$ and $\{q\}$, therefore the Module Theorem states that the only answer set of $(5.2) \sqcup (5.3)$ is $\{q, s\} \cup \{q\} = \{q, s\}$. This matches our previous analysis.

5.3 The Module Theorem for General Theory of Stable Models

As previously observed, the Splitting Theorem by Ferraris *et al.* and the Module Theorem by Janhunen *et al.* share a number of similarities both in form and function, however neither fully subsumes the other in terms of generality. In addition, despite their common ancestor, no effort has been made in order to relate these theorems.

We consider a first-order extension of the Module Theorem, which, when considering module components whose signatures are fixed to be the same, can be viewed as a generalization of Ferraris *et al.*'s Splitting Theorem.

In order to do this, we first consider a *partial interpretation* which fixes the evaluation of some subset of the signature. Formally, an interpretation of some signature $\mathbf{c}$ is a partial interpretation of the overarching signature σ if $\mathbf{c} \subseteq \sigma$.

We extend the notion of compatibility as follows: Given two partial interpretations I_1 and I_2 of signatures $\mathbf{c}_1$ and $\mathbf{c}_2$, respectively, we say I_1 and I_2 are *compatible* if, for each symbol $c \in (\mathbf{c}_1 \cap \mathbf{c}_2)$ $c^{I_1} = c^{I_2}$. Given two such compatible partial interpretations, we define their *union* $I_1 \cup I_2$ to be the interpretation of signature $\mathbf{c}_1 \cup \mathbf{c}_2$ such that, for each symbol $c \in \mathbf{c}_1 \cup \mathbf{c}_2$

- $c^{I_1} = c^{I_1 \cup I_2}$ if $c \in \mathbf{c}_1$, and
- $c^{I_2} = c^{I_1 \cup I_2}$ if $c \in \mathbf{c}_2$.

In the event that $\mathbf{c}_1$ and $\mathbf{c}_2$ are propositional, the definitions of compatibility between, and the union of, I_1 and I_2 correspond exactly to Janhunen *et al.*'s definition of compatibility and set union, respectively.

A *(first-order) module* $\mathbb{F}$ is a triple $\langle F, I, O\rangle$ where F is a first order sentence of σ and I and O are disjoint lists of predicate symbols of σ such that $Pred(F) \subseteq (I \cup O)$.

Similar to Janhunen *et al.*'s modules, a first-order module is essentially a formula which characterizes the behavior each of the predicates within its output given the value of each of the predicates in its input. Formally, this is characterized by treating each input predicate as non-intensional, which exempts it from minimality checking by the SM operator and essentially corresponds to choice rules (Ferraris *et al.* (2011)).

Definition 11 (Module Stable Models) *We say that some interpretation I is a* (module) stable model *of a first-order module $\mathbb{F} = \langle F, I, O\rangle$ if $I \models \mathrm{SM}[F; O]$. Additionally, we understand $\mathrm{SM}[\mathbb{F}]$ as shorthand for $\mathrm{SM}[F; O]$.*

We define the join of two first-order modules in the same way as the join between DLP-modules.

Definition 12 (Join of First-order Modules) *Given two first-order modules $\mathbb{F}_1 = \langle F_1, I_1, O_1\rangle$ and $\mathbb{F}_2 = \langle F_2, I_2, O_2\rangle$, they are called joinable if*

- *each strongly connected component of $\mathrm{DG}[F_1 \wedge F_2;\ O_1 \cup O_2]$ is either a subset of O_1 or a subset of O_2,*
- *any rule in F_1 with a strictly positive occurrence of a predicate in O_2 also occurs in F_2, and symmetrically*
- *any rule in F_2 with any strictly positive occurrence of a predicate in O_1 also occurs in F_1.*

Given two such first-order modules, the join of $\mathbb{F}_1$ and $\mathbb{F}_2$, denoted by $\mathbb{F}_1 \sqcup \mathbb{F}_2$, is defined as the first-order module

$$\langle F_1 \wedge F_2, (I_1 \cup I_2) \setminus (O_1 \cup O_2), O_1 \cup O_2\rangle$$

The following theorem is an extension of Theorem 2 to the general theory of Stable Models. Given a formula F, by $\mathbf{c}(F)$ we denote the set of all function and predicate constants occurring in F.

Theorem 3 (The Module Theorem for General Theory of Stable Models) *Let* $\mathbb{F}_1 = \langle F_1, I_1, O_1 \rangle$ *and* $\mathbb{F}_2 = \langle F_2, I_2, O_2 \rangle$ *be first-order modules that are joinable with interpretations* A_1 *and* A_2 *of* $\mathbf{c}_1 \supseteq \mathbf{c}(F_1) \cup O_1$ *and* $\mathbf{c}_2 \supseteq \mathbf{c}(F_2) \cup O_2$*, respectively. If* A_1 *and* A_2 *are compatible with each other,*

$$A_1 \cup A_2 \models \mathrm{SM}[\mathbb{F}_1 \sqcup \mathbb{F}_2] \quad \textit{iff} \quad A_1 \models \mathrm{SM}[\mathbb{F}_1] \ \textit{ and } \ A_2 \models \mathrm{SM}[\mathbb{F}_2] \ .$$

In the event $\sigma = \mathbf{c}_1 = \mathbf{c}_2$ and that F_1 and F_2 are negative on O_2 and O_2, respectively, Theorem 3 reduces to Theorem 1.

Furthermore, in the event that $\mathbf{c}_1$ and $\mathbf{c}_2$ are propositional, Theorem 3 reduces to Theorem 2 and can be used to trivially extend it to the case of arbitrary propositional formulas under the Stable Model Semantics.

5.4 Online Theories for Traditional Answer Set Programming

The systems iClingo (Gebser *et al.* (2008)) and oClingo (Gebser *et al.* (2011a)) extend the traditional answer set solving process by allowing the incremental instantiation and evaluation of an ASP program allowing for an efficient iterative deepening search without requiring multiple expensive restarts of the grounding/solving process. They divide a program into three template components used during the instantiation process: the *base component*, *incremental component*, and *volatile component*. oClingo takes this concept even further by allowing for online information to be dynamically added to the program during execution using an *online incremental component* and *online volatile component*.

$$\left\{ \begin{array}{c} (t \rightarrow n) \\ \wedge(q \wedge t \rightarrow p) \\ \wedge(r \wedge \neg s \rightarrow q) \\ \wedge(m \rightarrow r) \end{array} \right\} \underset{I=\{l,t\}}{\longmapsto} \left\langle \begin{array}{c} (t \rightarrow n) \\ \wedge(q \wedge t \rightarrow p) \end{array} , \{l,t\}, \{n,p\} \right\rangle$$

Figure 5.1: Module Instantiation of a Simple Traditional ASP Program

Intuitively, the *base component* contains static information, such as the initial state of a system; the *incremental component* contains step variant knowledge, such as how a system will evolve over time; and the *volatile component* contains information about the final step, such as the goal that should be reach (i.e. desirable attributes of the final state). In oClingo, the *online incremental component* contains information acquired during execution, while the *online volatile component* contains dynamic query information which can be used to provide temporary goals and similar information.

In this section, we review the background theories of oClingo.

Given a traditional ASP program F, the *projection* of F to a set of propositional atoms X, denoted $F|_X$, is defined to be the program obtained from F by removing all rules (2.1) in F that contain some a_i not in X such that $i > n$, and removing all occurrences of $\neg a_j$ $(1 \leq j \leq n)$ such that a_j is not in X from the remaining rules.

Definition 13 (Module Instantiation) *Given a traditional ASP program F, Gebser* et al. *(2011b) defined the module instantiation of F w.r.t. a set of propositional input atoms I, denoted $DLM(F, I)$, to be the module $(F|_{I \cup O}, I, O)$ where O is the set of all atoms occurring in the heads of rules in $F|_X$ and X is the set of all atoms in I or the heads of rules in F.*

The process of instantiating a module essentially models the simplification process that is performed by modern answer set solvers while they are grounding a program.

As an example, Figure 5.1 shows a simple program and the module which results from instantiating it with respect to $\{l, t\}$.

An *incrementally parametrized formula* $F[t]$ is a propositional formula which may contain *incrementally parametrized atoms* of the form $g(t)$:a where $g(t)$ is a meta-level function which maps $\mathbb{N}$ to $\mathbb{Z}$ and a is an atom.

Given any $k \in \mathbb{N}$ and incrementally parametrized formula $F[t]$, the *incremental instantiation* $F[t/k]$ is the formula which is obtained by replacing each incrementally parametrized atom $g(t)$:a with an atom v:a, where v is the result of evaluating $g(k)$. We adopt a similar notation for sets of incrementally parametrized atoms.

Example 9 (Incremental instantiation) *Consider the incremental parametrized formula*

$$((t-1)\text{:}p \wedge \neg(t-1)\text{:}q \rightarrow t\text{:}p \vee \neg t\text{:}p) \wedge ((t-1)\text{:}q \rightarrow t\text{:}r) \tag{5.5}$$

$(5.5)[t/1]$ *is then the formula*

$$(0\text{:}p \wedge \neg 0\text{:}q \rightarrow 1\text{:}p \vee \neg 1\text{:}p) \wedge (0\text{:}q \rightarrow 1\text{:}r)$$

Gebser *et al.* defined a *(traditional ASP) incremental theory* to be a triple $\langle B, P[t], Q[t]\rangle$ such that B is a traditional ASP program and $P[t]$ and $Q[t]$ are incrementally parametrized traditional ASP program. Informally, B is the *base component*, which describes static knowledge; $P[t]$ is the *cumulative component*, which contains information regarding every step that should be accumulated during execution; and $Q[t]$ is the *volatile component*, which contains constraints or other information regarding the final step.

A *(traditional ASP) online progression* $\langle E, F\rangle_{\geq 1}$ is a stream of pairs $(E_i[e_i], F_i[f_i])$ $(i \geq 1)$ of traditional ASP programs with associated non-negative integers e_i, f_i s.t. $f_i \geq e_i$ [2] . Intuitively, each $E_i[e_i]$ and $F_i[f_i]$ corresponds to stable and volatile knowl-

[2] Gebser *et al.* does not restrict the values of f_i and e_i, however we find this assumption simplifies the formalization without loss of generality as f_i can always be increased to match e_i without affecting the results.

edge acquired during execution, respectively. For each $(E_i[e_i], F_i[f_i])$, e_i and f_i denote the step for which they are relevant allowing knowledge to be acquired out of order.[3]

Given a traditional ASP incremental theory $\langle B, P[t], Q[t]\rangle$, online progression $\langle E, F\rangle_{\geq 1}$, and any $j, k \geq 0$ such that $e_1, \ldots, e_j, f_j \leq k$ the *incremental components* of the theory are

$$\{B, P[t/1], \ldots, P[t/k], E_1[e_1], \ldots, E_j[e_j], Q[t/k], F_j[f_j]\}^4 \tag{5.6}$$

We assume the presence of a set of atoms $I(F)$ for each $F \in (5.6)$ such that $I(F)$ does not contain any atoms within the head of any rule in F.

In addition, Gebser *et al.* defined the *k-expansion* to be the traditional ASP program $R_{j,k} = \bigwedge_{F \in (5.6)} F$.

Definition 14 (Modular Traditional ASP Theories) *Given a traditional ASP incremental theory $\langle F, P[t], Q[t]\rangle$ and traditional ASP online progression $\langle E, F\rangle_{\geq 1}$, they are modular if the following modules are well defined for all $j, k \geq 0$ such that $j, k \geq 0$ where $e_1, \ldots, e_j, f_j \leq k$* [5]*:*

$$\begin{aligned}
\mathbb{P}_0 &= DLM(B, I(B)), && \mathbb{E}_0 = \langle \top, \emptyset, \emptyset\rangle, \\
\mathbb{P}_i &= \mathbb{P}_{i-1} \sqcup DLM(P[t/i], O(\mathbb{P}_{i-1}) \cup I(P[t/i])), && (1 \leq i \leq k) \\
\mathbb{E}_i &= \mathbb{E}_{i-1} \sqcup DLM(E_i[e_i], O(\mathbb{P}_{e_i}) \cup O(\mathbb{E}_{i-1}) \cup I(E_i[e_i])) && (1 \leq i \leq j) \\
\mathbb{R}_{j,k} &= \mathbb{P}_k \sqcup \mathbb{E}_j \sqcup DLM(Q[k/t], O(\mathbb{P}_k) \cup I(Q[t/k])) \\
&\quad \sqcup DLM(F_j[f_j], O(\mathbb{P}_{f_j}) \cup O(\mathbb{E}_j) \cup I(F_j[f_j]))
\end{aligned}$$

[3] As an example, $E_4[3]$ is the 4th piece of online input and contains information relevant to step 3.

[4] For notational simplicity, we define $E_0[e_0]$ and $F_0[f_0]$ to be $\top$, e_0, f_0 to be 0, and $I(E_0[e_0])$ and $I(F_0[f_0])$ to be $\emptyset$.

[5] Given a module $\mathbb{F} = \langle F, I, O\rangle$, $O(\mathbb{F})$ refers to O and $I(\mathbb{F})$ refers to I.

We commonly refer to $\mathbb{R}_{j,k}$ as the *incremental composition* of the incremental theory and online progression.

Gebser *et al.* demonstrated that, given a modular traditional ASP incremental theory and online progression and some $j, k \geq 0$, we are able to evaluate each incremental component and compose the results in order to obtain the answer sets of the complete incremental composition $\mathbb{R}_{j,k}$ by applying Janhunen *et al.*'s Module Theorem repeatedly.

Proposition 5 (Composition of Solutions, Gebser *et al.* (2011b))
Given a modular traditional ASP incremental theory $\langle B, P[t], Q[t]\rangle$ and online progression $\langle E, F\rangle_{\geq 1}$ and any $j, k \geq 0$ such that $e_1, \ldots, e_j, f_j \leq k$, an interpretation I is an answer set of $\mathbb{R}_{j,k}$ iff there are compatible interpretations

$$I_B, I_{P[t/1]}, \ldots, I_{P[t/k]}, I_{E_1[e_1]}, \ldots, I_{E_j[e_j]}, I_{Q[t/k]}, I_{F_j[f_j]} \tag{5.7}$$

such that $I = \bigcup_{X \in (5.7)} X$ where

- *I_B is an answer set of $DLM(B, I(B))$,*
- *each $I_{P[t/i]}$ is an answer set of $DLM(P[t/i], O(\mathbb{P}_{i-1}) \cup I(P[t/i]))$,*
- *each $I_{E_i[e_i]}$ is an answer set of $DLM(E_i[e_i], O(\mathbb{P}_{e_i}) \cup O(\mathbb{E}_{i-1}) \cup I(E_i[e_i]))$,*
- *$I_{Q[t/k]}$ is an answer set of $DLM(Q[k/t], O(\mathbb{P}_k) \cup I(Q[t/k]))$, and*
- *$I_{F_j[f_j]}$ is an answer set of $DLM(F_j[f_j], O(\mathbb{P}_{f_j}) \cup O(\mathbb{E}_j) \cup I(F_j[f_j]))$.*

Given an incremental theory $\langle B, P[t], Q[t]\rangle$, online progression $\langle E, F\rangle_{\geq 1}$, we assume a precedence relationship $\prec$ on the elements in

$$\{B, P[t/1], P[t/2], \ldots, E_1[e_1], E_2[e_2], \ldots, Q[t/0], Q[t/1], \ldots, F_1[f_1], F_2[f_2], \ldots\} \tag{5.8}$$

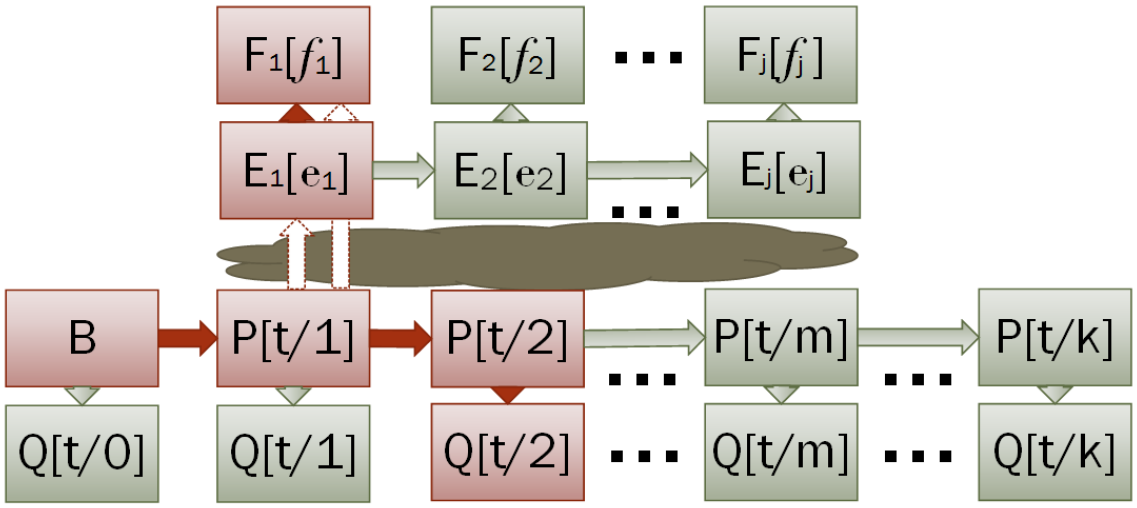

Figure 5.2: Precedence graph of component formulas

as the transitive closure of the following rules:

$$B \prec P[t/1] \prec \cdots \prec P[t/k], \qquad P[t/i] \prec Q[t/i], \qquad (i \geq 1)$$

$$E_1[e_1] \prec \cdots \prec E_j[e_j], \qquad E_i[e_i] \prec F_i[f_i], \qquad (i \geq 1)$$

$$P[t/e_i] \prec E_i[e_i], \text{ and} \qquad P[t/f_i] \prec F_i[f_i]. \qquad (1 \geq 1).$$

Additionally, we say that two formulas $F, G \in (5.8)$ *coexist* if there is some $j, k \in \mathbb{N}$ where $e_1, \ldots, e_j, f_j \leq k$ such that $F, G \in (5.6)$.

Intuitively, F, G coexist if they must eventually be composed together into some $\mathbb{R}_{j,k}$. For example, $P[t/1]$ and $Q[t/3]$ coexist as they are both present in $\mathbb{R}_{0,3}$, whereas $Q[t/1]$ and $Q[t/3]$ do not.

Definition 15 *We say that an incremental theory $\langle B, P[t], Q[t] \rangle$ and online progression $\langle E, F \rangle_{\geq 1}$ are mutually revisable if, for each pair of distinct coexisting formulas $F, G \in (5.8)$ such that $F \not\prec G$, it holds that each atom occurring in G which also occurs in the head of some rule in F is in $I(G)$.*

Technically, this definition is not as general as the one provided by Gebser *et al.* as they enforce that inputs of the instantiated module must be disjoint from the simplified formula's head atoms. We enforce this on the unsimplified formula as it provides a clearer definition. The cases which are affected by this change are those in which a rule initially contains a occurrence of an input atom in the head and is simplified out during the instantiation process.

For example, the incremental theory

$$\langle p \leftarrow q, \top, \top \rangle$$

such that $I(B) = \{p\}$ and $I(P[t/i]) = I(Q[t/i]) = \emptyset$ is mutually revisable by the original definition, but not by our altered definition.

Proposition 6 (Correctness, Gebser *et al.* (2011b)) *Given a traditional ASP incremental theory $\langle B, P[t], Q[t] \rangle$ and online progression $\langle E, F \rangle_{\geq 1}$ which are modular and mutually revisable, $j, k \geq 0$ such that $e_1, \ldots, e_j, f_j \leq k$. Let $R_{j,k}$ and $\mathbb{R}_{j,k} = \langle G, I, O \rangle$ be the k-expansion and incremental composition, respectively. It holds that the answer sets of $R_{j,k}$ and G coincide.*

Using Proposition 6 it is possible to incrementally ground, simplify, and solve a traditional ASP incremental theory in order to find the minimum k such that $R_{0,k}$ has an answer set without repeating previous work performed. In practice, this allows for a significant speedup when performing an iterative deepening search, such as when searching for a minimum length plan to accomplish a goal. In addition, the system is able to account for specific forms of online input in an equally efficient manner by allowing external information to be asserted in the online progression during execution.

Example 10 (Online ASP Solving) *Given an incremental theory*

$$\langle \top, \neg(t-1){:}q \land \neg(t-1){:}p \to t{:}p, \neg t{:}p \to \bot \rangle$$

such that $I(B) = \emptyset$, $I(P[t/i]) = \{(i-1){:}q\}$, *and* $I(Q[t/i]) = \emptyset$ *and online progression* $\langle E, F \rangle_{\geq 1}$.

Initially, $\mathbb{R}_{0,0}$ *is constructed such that*

$$\begin{aligned}
\mathbb{P}_0 &= \langle \top, \emptyset, \emptyset \rangle, && \textit{(trivially)} \\
\mathbb{Q}[0] &= DLM(Q[t/0], O(\mathbb{P}_0) \cup I(Q[t/0])) \\
&= \langle \top \to \bot, \emptyset, \emptyset \rangle, && \textit{(}\neg 0{:}p \textit{ is simplified to } \top\textit{)} \\
\mathbb{R}_{0,0} &= \mathbb{P}_0 \sqcup \mathbb{Q}[0] \\
&= \langle \top \to \bot, \emptyset, \emptyset \rangle.
\end{aligned}$$

It is easy to see that $\mathbb{R}_{0,0}$ *has no answer sets as* $\top \to \bot$ *is a contradiction.*

As a result, $\mathbb{R}_{0,1}$ *is attempted as follows:*

$$\begin{aligned}
\mathbb{P}_1 &= \mathbb{P}_0 \sqcup DLM(P[t/1], O(\mathbb{P}_0) \cup I(P[t/1])) \\
&= \mathbb{P}_0 \sqcup \langle \neg 0{:}q \to 1{:}p, \{0{:}q\}, \{1{:}p\} \rangle && \textit{(}\neg 0{:}p \textit{ is simplified to } \top\textit{)} \\
&= \langle \neg 0{:}q \to 1{:}p, \{0{:}q\}, \{1{:}p\} \rangle, \\
\mathbb{Q}[1] &= DLM(Q[t/1], O(\mathbb{P}_1) \cup I(Q[t/1])) \\
&= \langle \neg 1{:}p \to \bot, \{1{:}p\}, \emptyset \rangle, \\
\mathbb{R}_{0,1} &= \mathbb{P}_1 \sqcup \mathbb{Q}[1] \\
&= \langle (\neg 0{:}q \to 1{:}p) \land (\neg 1{:}p \to \bot), \{0{:}q\}, \{1{:}p\} \rangle.
\end{aligned}$$

Solving is then halted as this has one answer set $\{1{:}p\}$. *However, in the event that*

$$E_1[0] = 0{:}q, \textit{ and}$$

$$F_1[0] = \top$$

such that $I(E_1[0]) = I(F_1[0]) = \emptyset$ *is later asserted (i.e.* $0{:}q$ *must be true), we then must consider the construction of* $\mathbb{R}_{1,k}$*, rather than* $\mathbb{R}_{0,k}$*.*

$\mathbb{R}_{1,1}$ *is constructed such that*

$$\begin{aligned}
\mathbb{E}_1 &= \langle \top, \emptyset, \emptyset \rangle \sqcup DLM(E_1[0], O(\mathbb{P}_{e_0}) \cup I(E_1[0])) && (e_0 \text{ is } 0)\\
&= \langle \top, \emptyset, \emptyset \rangle \sqcup \langle 0{:}q, \emptyset, \{0{:}q\} \rangle \\
&= \langle 0{:}q, \emptyset, \{0{:}q\} \rangle, \\
\mathbb{F}[1] &= DLM(F_1[0], O(\mathbb{P}_{f_0}) \cup O(\mathbb{E}_1) \cup I(F_1[0])) && (f_1 \text{ is } 0)\\
&= \langle \top, \{0{:}q, 1{:}p\}, \emptyset \rangle, \\
\mathbb{R}_{1,1} &= \mathbb{P}_1 \sqcup \mathbb{E}_1 \sqcup \mathbb{Q}[1] \sqcup \mathbb{F}[1] \\
&= \langle (\neg 0{:}q \rightarrow 1{:}p) \wedge 0{:}q \wedge (\neg 1{:}p \rightarrow \bot), \{0{:}q, 1{:}p\} \rangle.
\end{aligned}$$

Once again, there are no answer sets, so the search is deepened to $\mathbb{R}_{1,2}$ *as follows:*

$$\begin{aligned}
\mathbb{P}_2 &= \mathbb{P}_1 \sqcup DLM(P[t/2], O(\mathbb{P}_1) \cup I(P[t/2])) \\
&= \mathbb{P}_1 \sqcup \langle \neg 1{:}q \wedge \neg 1{:}p \rightarrow 2{:}p, \{1{:}q, 1{:}p\}, \{2{:}p\} \rangle \\
&= \langle (\neg 0{:}q \rightarrow 1{:}p) \wedge (\neg 1{:}q \wedge \neg 1{:}p \rightarrow 2{:}p), \{0{:}q, 1{:}q\}, \{1{:}p, 2{:}p\} \rangle, \\
\mathbb{Q}[2] &= DLM(Q[t/2], O(\mathbb{P}_2) \cup I(Q[t/2])) \\
&= \langle \neg 2{:}p \rightarrow \bot, \{1{:}p, 2{:}p\}, \emptyset \rangle, \\
\mathbb{R}_{1,2} &= \mathbb{P}_2 \sqcup \mathbb{E}_1 \sqcup \mathbb{Q}[2] \sqcup \mathbb{F}[1] \\
&= \langle (\neg 0{:}q \rightarrow 1{:}p) \wedge (\neg 1{:}q \wedge \neg 1{:}p \rightarrow 2{:}p) \wedge 0{:}q \wedge (\neg 2{:}p \rightarrow \bot), \{1{:}q\}, \{0{:}q, 1{:}p, 2{:}p\} \rangle.
\end{aligned}$$

This has a single answer set $\{0{:}q, 2{:}p\}$*.*

The background theory of the system iClingo can be viewed as a special case of the oClingo theory such that the online progression $\langle E, F \rangle_{\geq 1}$ is trivial and, for each incremental component $F \in (5.6)$, the explicit inputs $I(F)$ are empty.

Chapter 6

REVISITING LANGUAGE $\mathcal{BC}+$ WITH ONLINE EXECUTION

Solving online planning problems consists of two major steps:

1. generate an optimal solution based on the current system state and known input, and
2. respond to new orders and other unexpected events during execution.

Traditionally, KR solutions to this involve attempting to solve a problem parametrized with a maximum step value k for each assignment $0, 1, \ldots$ of k until a solution is found. Then, if an unexpected event occurs during execution, this information regarding the event is added to the program and the process is repeated once again.

With this approach, if the minimum plan length is 10 steps, the system will re-evaluate the same program 10 times in order to generate the initial plan. If an event were to occur causing the minimum length of a plan to increase to 12, the system will then have re-evaluated the program 22 times in total. It is clear that this approach is not scalable and is therefore not appropriate for problems of non-trivial size or those which are time sensitive.

It is possible to use an incremental solver, such as oCLINGO to provide a more efficient alternative to this process, however Gebser *et al.*'s online ASP formalization relies on the conditions of modularity and mutual revisability, which are difficult to verify in practice. To make matters worse, the systems push the burden of checking these conditions to the developer, rather than performing automatic verification capable of detecting a violation of these conditions. This results in a prohibitive increase in developmental complexity.

Instead, we wish to be able to use a high-level language, such as $\mathcal{BC}+$, within their incremental theory in order to hide this complexity from the developer while taking advantageous of the sizable performance increase when considering planning problems.

In order to do this, we first present an extension of Gebser *et al.*'s online ASP formalization to allow for arbitrary propositional formulas under the Stable Model Semantics. Following this, we consider an extension to the syntax and semantics of $\mathcal{BC}+$ which allows for fault tolerant online execution under this framework. Finally, we observe that as a special case of these semantics, we can evaluate offline $\mathcal{BC}+$ programs incrementally, resulting in a significant improvement in performance when searching for minimum length plans.

6.1 Online Theories for Propositional ASP Formulas

We extend the notions of incremental theories and online progressions presented in Section 5.4 in a straight forward manner. That is, we define an *incremental theory* to be a triple $\langle B, P[t], Q[t] \rangle$ such that B is a propositional formula, and $P[t]$ and $Q[t]$ are incrementally parametrized formulas. An *online progression* $\langle E, F \rangle_{\geq 1}$ is a stream of pairs $(E_i[e_i], F_i[f_i])$ $(i \geq 1)$ of propositional formulas with associated non-negative integers e_i, f_i.

Similar to in Section 5.4, we assume the presence of a set of atoms $I(F)$ for each $F \in (5.6)$ such that $I(F)$ does not contain any atoms within the head of any rule in F.

Given a propositional formula F and set of atoms A, we define the projection of F onto A (denoted $F|_A$) to be the formula obtained by replacing all occurrences of atoms p in F such that $p \notin A$ with $\bot$ and performing the following syntactic

transformations recursively until no further transformations are possible:

$$\begin{array}{llll} \neg\bot \mapsto \top & \neg\top \mapsto \bot & & \\ \bot \wedge F \mapsto \bot & F \wedge \bot \mapsto \bot & \top \wedge F \mapsto F & F \wedge \top \mapsto F \\ \bot \vee F \mapsto F & F \vee \bot \mapsto F & \top \vee F \mapsto \top & F \vee \top \mapsto \top \\ \bot \rightarrow F \mapsto \top & F \rightarrow \top \mapsto \top & \top \rightarrow F \mapsto F & \end{array}$$

Formally, given a propositional formula F and some set of atoms A, we define $\mathit{Simple}(F, A)$ to be the results of recursively projecting F onto the set of atoms in A and the heads of rules within F until a fixpoint is reached.

Example 11 *As an example, take the propositional formula*

$$F = (p \rightarrow q) \wedge (q \rightarrow r) \wedge ((\neg q \wedge t) \rightarrow s).$$

The process of calculating $\mathit{Simple}(F, \{t, m\})$ *results in the following transformations on* F*:*

$$\begin{array}{ll} (p \rightarrow q) \wedge (q \rightarrow r) \wedge ((\neg q \wedge t) \rightarrow s) & \textit{initially} \\ \Rightarrow (q \rightarrow r) \wedge ((\neg q \wedge t) \rightarrow s) & \textit{first iteration} \\ \Rightarrow ((\neg q \wedge t) \rightarrow s) & \textit{second iteration} \\ \Rightarrow t \rightarrow s & \textit{third iteration} \\ \Rightarrow t \rightarrow s. & \textit{final iteration} \end{array}$$

In addition, we define the modular instantiation of F with respect to A, denoted $\mathit{PM}(F, A)$, to be the module

$$\langle \mathit{Simple}(F, A), A, \mathit{Pred}(\mathit{Simple}(F, A)) \setminus A \rangle.$$

When considering the common syntax, our modular instantiation and Gebser *et al.*'s are similar, except that we perform the projection simplification recursively to a fixpoint whereas they limit themselves to two iterations.

Definition 16 (Modular Incremental Theories and Online Progression) *Given an incremental theory* $\langle B, P[t], Q[t]\rangle$ *and online progression* $\langle E, F\rangle_{\geq 1}$ *we say they are modular if the following modules are defined for every* $j, k \in \mathbb{N}$ *such that* $e_1, \ldots, e_j, f_j \leq k$*:*[1]

$$\begin{aligned}
\mathbb{P}_0 &= PM(B, I(B)), & \mathbb{E}_0 &= \langle \top, \emptyset, \emptyset\rangle, \\
\mathbb{P}_i &= \mathbb{P}_{i-1} \sqcup PM(P[t/i], O(\mathbb{P}_{i-1}) \cup I(P[t/i])), & & (1 \leq i \leq k) \\
\mathbb{E}_i &= \mathbb{E}_{i-1} \sqcup PM(E_i[e_i], O(\mathbb{P}_{e_i}) \cup O(\mathbb{E}_{i-1}) \cup I(E_i[e_i])) & & (1 \leq i \leq j) \\
\mathbb{R}_{j,k} &= \mathbb{P}_k \sqcup \mathbb{E}_j \sqcup PM(Q[k/t], O(\mathbb{P}_k) \cup I(Q[t/k])) \\
&\quad \sqcup PM(F_j[f_j], O(\mathbb{P}_{f_j}) \cup O(\mathbb{E}_j) \cup I(F_j[f_j]))
\end{aligned}$$

Definition 16 is essentially the same as Definition 14, with the previously noted difference regarding the simplification performed during module instantiation. Due to this alternate simplification scheme, Definition 16 is a strict generalization of its counterpart, even when considering the common syntax.

As an example, consider the traditional ASP incremental theory and online progression

$$\begin{aligned}
&\langle r \to p, \top, (r \vee \neg r) \wedge (r \to p)\rangle \\
&E_1[0] = \top, \text{ and} \\
&F_1[0] = \top
\end{aligned}$$

such that $I(B) = \{r\}$, $I(Q[t/i]) = \{p\}$, and $I(P[t/i]) = I(E_1[1]) = I(F_1[1]) = \emptyset$. It holds that this is modular according to Definition 16, but not according to Definition 14.

[1] As in Section 5.4, we define $E_0[e_0]$ and $F_0[f_0]$ to be $\top$, e_0, f_0 to be 0, and $I(E_0[e_0])$ and $I(F_0[f_0])$ to be $\emptyset$.

We extend the notion of mutually revisable incremental theories and online progressions provided in Section 5.4 in a straight-forward manner.

Proposition 7 (Composition of Solutions) *Given a modular incremental theory* $\langle B, P[t], Q[t]\rangle$ *and online progression* $\langle E, F\rangle_{\geq 1}$ *and any* $j, k \geq 0$ *such that* $e_1, \ldots, e_j, f_j \leq k$*, an interpretation* I *is an answer set of* $\mathbb{R}_{j,k}$ *iff there are compatible interpretations*

$$I_B, I_{P[t/1]}, \ldots, I_{P[t/k]}, I_{E_1[e_1]}, \ldots, I_{E_j[e_j]}, I_{Q[t/k]}, I_{F_j[f_j]} \tag{6.1}$$

such that $I = \bigcup_{X \in (6.1)} X$ *where*

- I_B *is an answer set of* $PM(B, I(B))$*,*
- *each* $I_{P[t/i]}$ *is an answer set of* $PM(P[t/i], O(\mathbb{P}_{i-1}) \cup I(P[t/i]))$*,*
- *each* $I_{E_i[e_i]}$ *is an answer set of* $PM(E_i[e_i], O(\mathbb{P}_{e_i}) \cup O(\mathbb{E}_{i-1}) \cup I(E_i[e_i]))$*,*
- $I_{Q[t/k]}$ *is an answer set of* $PM(Q[k/t], O(\mathbb{P}_k) \cup I(Q[t/k]))$*, and*
- $I_{F_j[f_j]}$ *is an answer set of* $PM(F_j[f_j], O(\mathbb{P}_{f_j}) \cup O(\mathbb{E}_j) \cup I(F_j[f_j]))$*.*

Proposition 8 (Correctness of Incremental Composition) *Given a modular and mutually revisable incremental theory* $\langle B, P[t], Q[t]\rangle$ *and online progression* $\langle E, F\rangle_{\geq 1}$ *and some* $j, k \in \mathbb{N}$ *such that* $e_1, \ldots, e_j, f_j \leq k$ *and let* $R_{j,k}$ *and* $\mathbb{R}_{j,k} = \langle G, I, O\rangle$ *be the* k*-expansion and incremental composition of the incremental theory and online progression. The answer sets of* $R_{j,k}$ *and* G *coincide.*

As it happens, in the event that all explicit inputs are empty (i.e. the offline case) mutually revisability is a stronger condition than modularity. This means that in offline environments it's sufficient to just check that an incremental theory is mutually revisable.

In practice, these conditions are quite difficult to check as they require considering complex relationships between all current and future incremental components

and their modular instantiations. As it turns out, it is possible to strengthen these conditions to simplify them greatly. Definition 17 provides such a strengthening in the form of *acyclic* incremental theories and online progressions. Meanwhile, Lemma 1 provides that this condition is sufficient to check for modularity and mutual revisability.

Definition 17 (Acyclic Incremental Theories and Online Progressions) *Given an incremental theory $\langle B, P[t], Q[t] \rangle$ and online progression $\langle E, F \rangle_{\geq 1}$, we say they are acyclic if:*

- *for each distinct coexisting $F, G \in (5.8)$ such that $F \not\prec G$, each occurrence of an atom within $Pred(G) \setminus I(G)$ in F is within a subformula of the form $\neg H$, and*
- *for each $F \in (5.8)$, each occurrence of an atom within $I(F)$ in F is within a subformula of the form $\neg H$.*

Lemma 1 (Acyclic Modularity and Mutual Revisability) *Given an acyclic incremental theory $\langle B, P[t], Q[t] \rangle$ and online progression $\langle E, F \rangle_{\geq 1}$, it holds that they are modular and mutually revisable.*

Although the acyclic condition loses some generality over modularity and mutual revisability, it provides several practical advantages over them in that it avoids referring to the module instantiation process and dependency graphs, and instead provides a simple syntactic condition to consider.

6.1.1 Reducing Online Propositional ASP Theories to Disjunctive Logic Theories

Given an incremental ASP theory and online progression, we have shown how the theory may be considered incrementally in a manner similar to that provided by the systems iClingo and oClingo for traditional incremental ASP theories. Our

incremental ASP theory and those used by these systems correspond for the common syntax. In this section, we show how a larger subset of our incremental ASP theories can be computed using these systems by applying the translation provided by Lee and Palla (2007) which is used as the basis of system F2LP (Lee and Palla (2009)).

Given any propositional formula F, the disjunctive logic formula *prop2dlf*(F) obtained by recursively performing the following strongly equivalent transformations on subformulas of F in the order they appear:[2]

$$\neg\top \quad \mapsto \quad \bot \tag{6.2}$$
$$\neg\bot \quad \mapsto \quad \top \tag{6.3}$$
$$\neg\neg\neg G \quad \mapsto \quad \neg G \tag{6.4}$$
$$\neg(G_1 \vee G_2) \quad \mapsto \quad \neg G_1 \wedge \neg G_2 \tag{6.5}$$
$$\neg(G_1 \wedge G_2) \quad \mapsto \quad \neg G_1 \vee \neg G_2 \tag{6.6}$$
$$\neg(G \rightarrow H) \quad \mapsto \quad \neg\neg G \wedge \neg H \tag{6.7}$$
$$\top \wedge G \quad \mapsto \quad G \tag{6.8}$$
$$\bot \wedge G \quad \mapsto \quad \bot \tag{6.9}$$
$$\top \vee G \quad \mapsto \quad \top \tag{6.10}$$
$$\bot \vee G \quad \mapsto \quad G \tag{6.11}$$

$$\top \rightarrow G \quad \mapsto \quad G \tag{6.12}$$
$$\bot \rightarrow G \quad \mapsto \quad \top \tag{6.13}$$
$$\neg\neg G_1 \wedge G_2 \rightarrow H \quad \mapsto \quad G_2 \rightarrow \neg G_1 \vee H \tag{6.14}$$
$$(G_1 \vee G_2) \wedge G_3 \rightarrow H \quad \mapsto \quad \bigwedge \left\{ \begin{array}{l} G_1 \wedge G_3 \rightarrow H \\ G_2 \wedge G_3 \rightarrow H \end{array} \right. \tag{6.15}$$
$$(G_1 \rightarrow G_2) \wedge G_3 \rightarrow H \quad \mapsto \quad \bigwedge \left\{ \begin{array}{l} \neg G_1 \wedge G_3 \rightarrow H \\ G_2 \wedge G_3 \rightarrow H \\ G_3 \rightarrow G_1 \vee \neg G_2 \vee H \end{array} \right. \tag{6.16}$$

[2] For clarity, we treat implication and negation separately here.

$$
\begin{array}{lcll}
G \to \top \vee H & \mapsto & \top & (6.17)\\
G \to \bot \vee H & \mapsto & \bot & (6.18)\\
G \to \neg\neg H_1 \vee H_2 & \mapsto & G \wedge \neg H_1 \to H_2 & (6.19)\\
G \to (H_1 \wedge H_2) \vee H_3 & \mapsto & \bigwedge \left\{ \begin{array}{l} G \to H_1 \vee H_3 \\ G \to H_2 \vee H_3 \end{array} \right. & (6.20)\\
G \to (H_1 \to H_2) \vee H_3 & \mapsto & \bigwedge \left\{ \begin{array}{l} G \wedge H_1 \to H_2 \\ G \wedge \neg H_2 \to \neg H_1 \vee H_3 \end{array} \right. & (6.21)
\end{array}
$$

As an example, consider the propositional formula

$$((p \to q) \wedge \neg\neg r) \vee \neg\neg\neg s \to t, \tag{6.22}$$

it is reduced to a disjunctive logic formula using the following transformations:

$$
\begin{array}{ll}
((p \to q) \wedge \neg\neg r) \vee \neg\neg\neg s \to t & \text{initially}\\
((p \to q) \wedge \neg\neg r) \vee \neg s \to t & (6.4)\\
\bigwedge \left\{ \begin{array}{l} (p \to q) \wedge \neg\neg r \to t \\ \neg s \to t \end{array} \right. & (6.15)\\
\bigwedge \left\{ \begin{array}{l} (p \to q) \to \neg r \vee t \\ \neg s \to t \end{array} \right. & (6.14)\\
\bigwedge \left\{ \begin{array}{l} \neg p \to \neg r \vee t \\ q \to \neg r \vee t \\ p \vee \neg q \vee \neg r \vee t \\ \neg s \to t \end{array} \right. & (6.16)
\end{array}
\tag{6.23}
$$

The result is a disjunctive logic formula which may have negation in the head of its rules. Alternatively, negated formulas in the head of a rule may be shifted into the body under double negation.

As an example, (6.23) may be equivalently represented as

$$
\begin{aligned}
\neg p \wedge \neg\neg r &\to t\\
q \wedge \neg\neg r &\to t\\
\neg\neg q \wedge \neg\neg r &\to p \vee t\\
\neg s &\to t.
\end{aligned}
$$

In the event the input formula is definite and each occurrence of implication is

- does not occur within the antecedent of another implication
- of the form $\neg H$, or
- occurs within a subformula $\neg H$,

the resulting formula is then a traditional ASP formula with double negation[3].

Although this transformation produces a disjunctive logic formula (or traditional ASP formula) that is strongly equivalent to the original, it does not preserve the dependency graph or strictly positive occurrences of atoms within the formula. For example, (6.22) is negative on p while *prop2dlf*((6.22)) is not.

We extend the notion of *prop2dlp* to be applicable to incrementally parametrized formulas in a straight forward manner.

Definition 18 (Compiling Incremental Theories into Disjunctive Logic) *Given an incremental theory $\langle B, P[t], Q[t]\rangle$ and online progression $\langle E, F\rangle_{\geq 1}$, we define the disjunctive logic compilations prop2dlf$(\langle B, P[t], Q[t]\rangle)$ and prop2dlf$(\langle E, F\rangle_{\geq 1})$ to be the incremental theory and online progression obtained by applying prop2dlf to B, $P[t]$, $Q[t]$, and each $E_i[e_i]$ and $F_i[f_i]$ $(i \geq 1)$.*

It is quite clear that this transformation does not preserve the modularity and mutual revisability of the incremental theory and online progression. As an example, consider the incremental theory

$$\langle p, \top, p \rightarrow q \rightarrow r\rangle.$$

[3] Double negation can then be simulated by introducing additional constants in the signature.

such that the online progression and all explicit inputs are empty. It holds that this incremental theory and online progression are modular and mutually revisable. However, *prop2dlf*$(\langle B, P[t], Q[t]\rangle)$ is

$$\langle p, \top, (\neg p \rightarrow r) \wedge (q \rightarrow r) \wedge (p \vee \neg q \vee r)\rangle$$

which is neither mutually revisable nor modular.

However, as is shown by Proposition 9, it turns out that acyclicity is preserved by the transformation. This allows us to capture a large class of propositional incremental theories within Gebser *et al.*'s disjunctive logic theories.

Proposition 9 (Compiling Acyclic Incremental Theories) *Given some incremental theory* $\langle B, P[t], Q[t]\rangle$ *and online progression* $\langle E, F\rangle_{\geq 1}$ *which are acyclic. The disjunctive logic compilation of* $\langle B, P[t], Q[t]\rangle$ *and* $\langle E, F\rangle_{\geq 1}$ *are also acyclic.*

Ultimately, proposition 9 allows us to compute a sizable fragment of modular and mutually revisable incremental theories using the existing systems iClingo and oClingo.

6.2 Defining the Online Action Language $\mathcal{BC}+$

Given the generalized form of Gebser *et al.* (2011a)'s online ASP theory, it then becomes possible to embed the semantics of $\mathcal{BC}+$ directly within it in order to take advantage of the incremental computational of systems such as iClingo. However, this still does not address the problem of handling exceptions which may occur during plan execution. In order to do this, we propose a proper extension $\mathcal{BC}+$ which leverages our online ASP theory in order to allow for knowledge to be passed into the program during plan execution in the form of external constants.

We assume the presence of a multi-valued signature σ partitioned similar to the signature described in Chapter 4 except with additional sets σ_{EF} and σ_{EA} of *external*

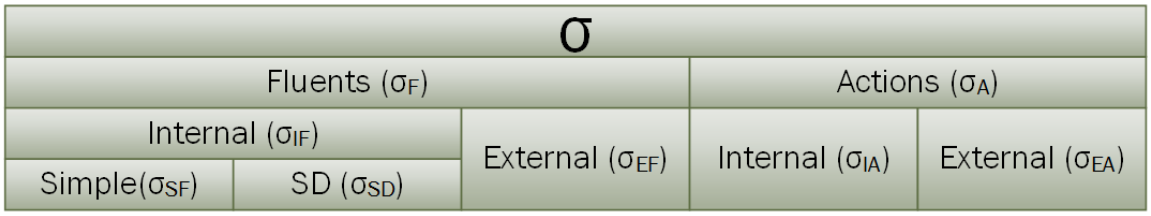

Figure 6.1: Hierarchy of symbols within a $\mathcal{BC}+$ signature

fluents and *external actions* such that $\sigma_{\mathrm{EF}} \subseteq \sigma_{\mathrm{F}}$ is disjoint from both σ_{SF} and σ_{SD} and $\sigma_{\mathrm{EA}} \subseteq \sigma_{\mathrm{A}}$. We recognize any fluent which is not an external fluent as an internal fluent, and likewise, any action which is not an external action is an internal action. The final result is a signature partitioned as in Figure 6.1.

In addition, we assume that the domain of each external fluent and each external action contains a special element `u`. Which, intuitively, corresponds to an unknown value.

We define an Online $\mathcal{BC}+$ ($o\mathcal{BC}+$) $\mathcal{D}$ action description to be a $\mathcal{BC}+$ action description which may contain external fluents and actions in the bodies of any law and does not contain these constant in the head of any law.

An *observation* is an expression of the form

$$\textbf{observed } c = v \textbf{ at } m \tag{6.24}$$

where $c = v$ is a multi-valued atom such that c is an external fluent or external action, $v \neq \mathtt{u}$, and $m \in \mathbb{N}$.

A *(observational) constraint* is an expression of the form

$$\textbf{constraint } F \textbf{ at } m \tag{6.25}$$

where F is a multi-valued propositional formula containing no external constants and $m \in \mathbb{N}$.

We say an observation (6.24) or observational constraint (6.25) is *dynamic* if it contains some action constant $a \in \sigma$, otherwise we say it is *static*.

We define an *observation stream* $\mathcal{O}_{n,\tilde{m}}$ to be a list

$$[(O_1, m_1), \ldots, (O_n, m_n)]$$

such that

- for each $(1 \leq i \leq n)$ O_i is a finite set of observations (6.24) and observational constraints (6.25), and m_i is the maximum of each m among the static observations and constraints and $m + 1$ among the dynamic observations and constraints,
- $\tilde{m}$ is the maximum of each m_i $(1 \leq i \leq n)$, and
- for each external constant $c \in \sigma$ and each $(1 \leq m \leq \tilde{m})$ it holds that there is at most one (6.24) in $O_1 \cup \cdots \cup O_n$.

Intuitively, observations are non-monotonic observations the agent has made regarding the defined external actions and fluents. Meanwhile, the observational constraints serve to further limit past histories according to what the agent knows, such as what actions the agent has executed.

Example 12 *As a simple example, consider an elaboration to the light switch problem considered in Chapter 3 where the light bulb may be burnt out. In the event this is the case, the light will not turn on until the bulb is replaced. This problem can be formalized in o$\mathcal{BC}$+ as is shown in Figure 6.2. Intuitively, Fault is the agent's internal model of whether the light is burnt out, while ExtFault represents the agent's external observations. Typically, Fault is ruled by inertia. However, in the event the agent gains additional information (i.e. observes whether there has been a fault) Fault is updated to reflect this. Performing ReplaceBulb will then reset the agent's internal model and the agent once again assumes that the fault has been fixed.*

Named Sets:	Value:		
Status	$\{\texttt{on}, \texttt{off}\}$		
Boolean	$\{\texttt{t}, \texttt{f}\}$		
ExtBoolean	$\{\texttt{t}, \texttt{f}, \texttt{u}\}$		
Constants:	**Type:**		**Domain:**
Sw	Simple Fluent		*Status*
Light	SD Fluent		*Status*
Flip	Action		*Boolean*
Fault	Simple Fluent		*ExtBoolean*
ExtFault	External Fluent		*ExtBoolean*
ReplaceBulb	Action		*Boolean*
inertial *Sw*.			$\mathcal{D}^{o\mathcal{BC}+}_{\text{switch},1}$
exogenous *Flip*.			$\mathcal{D}^{o\mathcal{BC}+}_{\text{switch},2}$
Flip = t **causes** *Sw* = on **if** *Sw* = off.			$\mathcal{D}^{o\mathcal{BC}+}_{\text{switch},3}$
Flip = t **causes** *Sw* = off **if** *Sw* = on.			$\mathcal{D}^{o\mathcal{BC}+}_{\text{switch},4}$
default *Light* = *s* **if** *Sw* = *s*.		$s \in$ *Status*	$\mathcal{D}^{o\mathcal{BC}+}_{\text{switch},5}$
inertial *Fault* **after** *ReplaceBulb* = f.			$\mathcal{D}^{o\mathcal{BC}+}_{\text{switch},6}$
exogenous *ReplaceBulb*.			$\mathcal{D}^{o\mathcal{BC}+}_{\text{switch},7}$
nonexecutable *ReplaceBulb* = t **if** *Flip* = t.			$\mathcal{D}^{o\mathcal{BC}+}_{\text{switch},8}$
Fault = *v* **if** *ExtFault* = *v*.		$v \in$ *Boolean*	$\mathcal{D}^{o\mathcal{BC}+}_{\text{switch},9}$
Light = off **if** *Fault* = t.			$\mathcal{D}^{o\mathcal{BC}+}_{\text{switch},10}$
ReplaceBulb = t **causes** *Choice*(*Fault* = u).			$\mathcal{D}^{o\mathcal{BC}+}_{\text{switch},11}$

Figure 6.2: Online Faulty Switch Elaboration in $o\mathcal{BC}+$.

6.3 Transition Systems for $o\mathcal{BC}+$

We extend the notion of $\mathcal{BC}+$ transition systems as follows: Given an Online $\mathcal{BC}+$ action description $\mathcal{D}$ we construct the transition system $\mathcal{T}(\mathcal{D})$ corresponding to the action description similar to the process described in Section 4.2. Formally,

- a *state* of a $\mathcal{T}(\mathcal{D})$ is an interpretation of σ_{F} which is an answer set of

$$(4.3) \wedge \mathit{Choice}(\sigma_{\text{EF}});$$

- given such a state $\mathcal{S}$, a *candidate transition label* of $\mathcal{T}(\mathcal{D})$ leaving $\mathcal{S}$ is an interpretation of σ_{A} such that $\mathcal{S} \cup \mathcal{A}$ is an answer set of

$$(4.4) \wedge \mathit{Choice}(\sigma_{\text{EA}});$$

- given states $\mathcal{S}$ and $\mathcal{S}'$ and a candidate transition label $\mathcal{A}$, $\langle \mathcal{S}, \mathcal{A}, \mathcal{S}' \rangle$ is a *transition* of $\mathcal{T}(\mathcal{D})$ if $\mathcal{S} \cup \mathcal{A} \cup \mathcal{S}'$ is an answer set of

$$(4.5) \wedge \mathit{Choice}(1{:}\sigma_{\text{EF}}).$$

It is clear that in the event that there are no external fluents or external actions in σ this definition is equivalent to the one provided in Section 4.1.

Given an observation stream $\mathcal{O}_{n,\tilde{m}}$ and history $\mathcal{H}_k$ such that $k \geq \tilde{m}$, we say that $\mathcal{H}_k$ observes $\mathcal{O}_{n,\tilde{m}}$ if, for each observation (6.24) in $\mathcal{O}_{n,\tilde{m}}$, it holds that $\mathcal{H}_k \models m{:}c = v$, and, for each constraint (6.25) in $\mathcal{O}_{n,\tilde{m}}$, it holds that $\mathcal{H}_k \models m{:}F$.

Typically, given a action description $\mathcal{D}$ and online progression $\mathcal{O}_{n,\tilde{m}}$ we assume that any future external constants can take any value arbitrarily. Often, it is useful to minimize a set A of these external constants, such as if they represent abnormal behavior in the system. In order to do this, we refer to a history which is *A-normal* with respect to $\mathcal{O}_{n,\tilde{m}}$. Intuitively, an A-normal history is one which we enforce that in the absence of knowledge regarding the value of any $c \in A$, c is $\mathtt{u}$.

Formally, given a set of external constants A and online progression $\mathcal{O}_{n,\tilde{m}}$, we say that a history $\mathcal{H}_k$ is *A-normal* with respect to $\mathcal{O}_{n,\tilde{m}}$, if, for each $c \in A$ and each $i \in \{0, \ldots, k\}$ such that $i \neq k$ if c is an action, it holds that if c does not occur in some observation

$$\textbf{observed } c = v \textbf{ at } m \qquad (6.24)$$

in $\mathcal{O}_{n,\tilde{m}}$ such that $m = i$, then $\mathcal{H}_k \models (i{:}c = \mathtt{u})$.

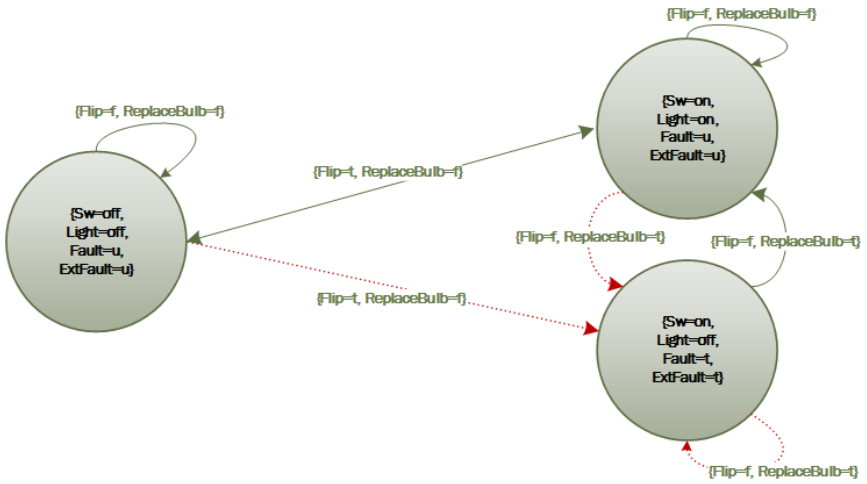

Figure 6.3: A Partial Transition System of $\mathcal{D}^{o\mathcal{BC}+}_{\text{switch}}$

Example 13 (Online $\mathcal{BC}+$ Transition Systems) *Consider the transition system corresponding to the toggle switch elaboration. The minimum length history from*

$$\mathcal{S}_0 = \{Switch = \texttt{off}, Light = \texttt{off}, Fault = \texttt{u}, ExtFault = \texttt{u}\}$$

to a state $\mathcal{S}$ such that $\mathcal{S} \models Light = \texttt{on}$ are $\mathcal{H}_{1,1} = [\mathcal{S}_0, \mathcal{A}_0, \mathcal{S}_1]$, and $\mathcal{H}_{1,2} = [\mathcal{S}_0, \mathcal{A}_0, \mathcal{S}_2]$ where

$$\mathcal{S}_1 = \{Switch = \texttt{on}, Light = \texttt{on}, Fault = \texttt{u}, ExtFault = \texttt{u}\},$$
$$\mathcal{S}_2 = \{Switch = \texttt{on}, Light = \texttt{on}, Fault = \texttt{f}, ExtFault = \texttt{f}\}, \text{ and}$$
$$\mathcal{A}_0 = \{Flip = \texttt{t}, ReplaceBulb = \texttt{f}\}.$$

Intuitively, the difference between $\mathcal{S}_1$ and $\mathcal{S}_2$ is that in $\mathcal{S}_1$ the agent has no knowledge as to whether a fault has occurred (i.e. the bulb has burnt out) whereas in $\mathcal{S}_2$ the agent knows that the light is fine.

Of the two, only $\mathcal{H}_{1,1}$ is $\{ExtFault\}$-normal with respect to the online progression $\mathcal{O}_{0,0} = []$.

If, following the execution of Flip, the agent observes that a fault did occur the knowledge can be added to the online progression producing

$$\mathcal{O}_{1,1} = [(\{\textbf{\textit{observed}}\ \mathit{ExtFault} = \mathtt{t}\ \textbf{\textit{at}}\ 1,\ \textbf{\textit{constraint}}\ \mathit{Flip} = \mathtt{t}\ \textbf{\textit{at}}\ 0\}, 1)].$$

(The addition of the constraint enforces that the agent has executed $\mathit{Flip} = \mathtt{t}$ *and prevents that action from being revised.) The new minimum length history which is* $\{\mathit{ExtFault}\}$*-normal to* $\mathcal{O}_{1,1}$ *is* $\mathcal{H}_2 = [\mathcal{S}_0, \mathcal{A}_0, \mathcal{S}_3, \mathcal{A}_1, \mathcal{S}_1]$ *where*

$$\mathcal{S}_3 = \{\mathit{Switch} = \mathtt{on}, \mathit{Light} = \mathtt{off}, \mathit{Fault} = \mathtt{t}, \mathit{ExtFault} = \mathtt{t}\},\ \textit{and}$$
$$\mathcal{A}_1 = \{\mathit{Flip} = \mathtt{f}, \mathit{ReplaceBulb} = \mathtt{t}\}.$$

This history essentially prescribes that the agent should replace the light bulb in order to attempt to fix the fault.

A partial specification of the transition system $\mathcal{T}(\mathcal{D}^{o\mathcal{BC}+}_{switch})$ *is shown in Figure 6.3. The dashed edges (in red) depend on the assertion of an external constant and therefore are not considered for future transitions in* $\{\mathit{ExtFault}\}$*-normal histories.*

6.4 Incremental Assembly of Online $\mathcal{BC}+$ Descriptions

Given a problem formalized in $o\mathcal{BC}+$, we are able to evaluate it using our online ASP theory with systems such as oClingo. This is advantageous over providing a native problem specification in a number of ways. First, as in offline $\mathcal{BC}+$, online $\mathcal{BC}+$ allows for a higher-level specification for problems which is intuitive without direct knowledge of the semantics. Second, the language is homogenous regardless of the execution platform being used and does not require any additional knowledge to take advantageous of the considerable performance increase garnered from using an incremental solving technique (even in an offline environment). Finally, as we will see in this section, the reduction into our online ASP theory is guaranteed to produce an

incremental theory and online progression which satisfy the conditions of modularity and mutual revisability, relieving the developer from considering these conditions.

Given an Online $\mathcal{BC}+$ action description $\mathcal{D}$, observation stream $\mathcal{O}_{n,\tilde{m}}$, and some incrementally parametrized multi-valued formula $\mathcal{Q}[t]$, we define the corresponding incremental theory $\langle B, P[t], Q[t]\rangle^{\mathcal{D},\mathcal{Q}[t]}$ and online progression $\langle E, F\rangle_{\geq 1}^{\mathcal{O}_{n,\tilde{m}}}$ as follows:

$$B = \bigwedge \begin{cases} 0{:}Choice(f) & \text{for each simple fluent } f \\ 0{:}G \to 0{:}F & \text{for each static law (4.1)} \\ 0{:}Choice(f = \mathtt{u}) & \text{for each external fluent } f \\ 0{:}UEC(\sigma_F) & \end{cases}$$

$$P[t] = \bigwedge \begin{cases} t{:}G \to t{:}F & \text{for each static law (3.13)} \\ (t-1){:}G \to (t-1){:}F & \text{for each action dynamic law (4.1)} \\ (t-1){:}H \wedge t{:}G \to t{:}F & \text{for each fluent dynamic law (4.2)} \\ t{:}Choice(f = \mathtt{u}) & \text{for each external fluent } f \\ (t-1){:}Choice(a = \mathtt{u}) & \text{for each external action } a \\ t{:}UEC(\sigma_F) & \\ (t-1){:}UEC(\sigma_A) & \end{cases}$$

$$Q[t] = \neg\neg\mathcal{Q}[t]$$

$$E_i[m_i] = \bigwedge \begin{cases} m{:}c = v & \text{for each observation (6.24)} \in O_i \\ \neg\neg m{:}F & \text{for each constraint (6.25)} \in O_i \end{cases}$$

$$F_i[m_i] = \top$$

Given a multi-valued propositional signature σ we define $At(\sigma)$ to be the set of multi-valued atoms $c = v$ where $c \in \sigma$ and $v \in Dom(c)$. Furthermore, we define $At_u(\sigma)$ to be the set of all such atoms such that $v \neq \mathtt{u}$.

We define the sets of explicit inputs as follows:

- $I(B) = At_u(0{:}\sigma_{\mathrm{EF}})$,
- $I(P[t/i]) = At_u(i{:}\sigma_{\mathrm{EF}} \cup (i-1){:}\sigma_{\mathrm{EA}})$,
- $I(Q[t/i]) = At_u(\bigcup_{0 \leq j < i}(j{:}\sigma_{\mathrm{EF}} \cup j{:}\sigma_{\mathrm{EA}}) \cup i{:}\sigma_{\mathrm{EF}})$, and

- $I(E_i) = I(F_i) = \emptyset$.

Proposition 10 and Corollary 2 provide that Online $\mathcal{BC}+$ (and, by extension, offline $\mathcal{BC}+$) results in an incremental theory and online progression which is able to be computed using the online ASP theory discussed in Section 6.1.

Proposition 10 (Modular and Mutually Revisable Construction) *Given an online $\mathcal{BC}+$ action description $\mathcal{D}$, observation stream $\mathcal{O}_{n,\tilde{m}}$, and some incrementally parametrized multi-valued formula $\mathcal{Q}[t]$, the incremental theory $\langle B, P[t], Q[t]\rangle^{\mathcal{D},\mathcal{Q}[t]}$ and online progression $\langle E, F\rangle_{\geq 1}^{\mathcal{O}_{n,\tilde{m}}}$ is modular and mutually revisable.*

Corollary 2 (Correctness of Incremental Assembly) *Given an online $\mathcal{BC}+$ action description $\mathcal{D}$, observation stream $\mathcal{O}_{n,\tilde{m}}$, some incrementally parametrized multi-valued formula $\mathcal{Q}[t]$, and some $k \geq \tilde{m}$. Let $\mathbb{R}_{\tilde{m},k} = \langle F, I, O\rangle$ be the modular composition of $\langle B, P[t], Q[t]\rangle^{\mathcal{D},\mathcal{Q}[t]}$ and $\langle E, F\rangle_{\geq 1}^{\mathcal{O}_{n,\tilde{m}}}$. The answer sets of F correspond to the histories of $\mathcal{T}(\mathcal{D})$ which observe $\mathcal{O}_{n,\tilde{m}}$, are $\sigma_{EF} \cup \sigma_{EA}$-normal with respect to $\mathcal{O}_{n,\tilde{m}}$, and satisfy $\mathcal{Q}[t/k]$.*

Example 14 (Continuation of Example 13) *Consider the light switch domain described in Example 13 and let $\mathcal{Q}[t]$ be*

$$0{:}Switch = \texttt{off} \wedge t{:}Light = \texttt{on}$$

which requests solutions in which the initial state's switch (and, by extension, light) is off and the final state's light is on. The domain description $\langle B, P[t], Q[t]\rangle^{\mathcal{D}_{switch}^{o\mathcal{BC}+},\mathcal{Q}[t]}$ is:

$$
B = \bigwedge \left\{ \begin{array}{ll}
0{:}\mathit{Choice}(\mathit{Switch} = \texttt{on}) \wedge 0{:}\mathit{Choice}(\mathit{Switch} = \texttt{off}) & \\
\bigwedge_{s \in \{\texttt{on},\texttt{off}\}} 0{:}\mathit{Switch} = s \rightarrow 0{:}\mathit{Choice}(\mathit{Light} = s) & \mathcal{D}^{o\mathcal{BC}+}_{switch,5} \\
0{:}\mathit{Fault} = \texttt{t} \rightarrow 0{:}\mathit{Light} = \texttt{off} & \mathcal{D}^{o\mathcal{BC}+}_{switch,10} \\
0{:}\mathit{Choice}(\mathit{Fault} = \texttt{u}) & \sigma_{EF} \\
0{:}\mathit{UEC}(\mathit{Switch}) \wedge 0{:}\mathit{UEC}(\mathit{Light}) \wedge 0{:}\mathit{UEC}(\mathit{Fault}) & \mathit{UEC}(\sigma_F)
\end{array} \right.
$$

$$
P[t] = \bigwedge \left\{ \begin{array}{ll}
\bigwedge_{s \in \{\texttt{on},\texttt{off}\}} t{:}\mathit{Switch} = s \rightarrow t{:}\mathit{Choice}(\mathit{Light} = s) & \mathcal{D}^{o\mathcal{BC}+}_{switch,5} \\
t{:}\mathit{Fault} = \texttt{t} \rightarrow t{:}\mathit{Light} = \texttt{off} & \mathcal{D}^{o\mathcal{BC}+}_{switch,10} \\
\bigwedge_{b \in \{\texttt{t},\texttt{f}\}} (t-1){:}\mathit{Choice}(\mathit{Flip} = b) & \mathcal{D}^{o\mathcal{BC}+}_{switch,2} \\
\bigwedge_{b \in \{\texttt{t},\texttt{f}\}} (t-1){:}\mathit{Choice}(\mathit{ReplaceBulb} = b) & \mathcal{D}^{o\mathcal{BC}+}_{switch,7} \\
(t-1){:}\mathit{Flip} = \texttt{t} \wedge (t-1) : \mathit{ReplaceBulb} = \texttt{t} \rightarrow \bot & \mathcal{D}^{o\mathcal{BC}+}_{switch,8} \\
\bigwedge_{s \in \{\texttt{on},\texttt{off}\}} (t-1){:}\mathit{Switch} = s \rightarrow t{:}\mathit{Choice}(\mathit{Switch} = s) & \mathcal{D}^{o\mathcal{BC}+}_{switch,1} \\
(t-1){:}\mathit{Flip} = \texttt{t} \wedge (t-1){:}\mathit{Switch} = \texttt{off} \rightarrow t{:}\mathit{Switch} = \texttt{on} & \mathcal{D}^{o\mathcal{BC}+}_{switch,3} \\
(t-1){:}\mathit{Flip} = \texttt{t} \wedge (t-1){:}\mathit{Switch} = \texttt{on} \rightarrow t{:}\mathit{Switch} = \texttt{off} & \mathcal{D}^{o\mathcal{BC}+}_{switch,4} \\
\bigwedge_{e \in \{\texttt{t},\texttt{f},\texttt{u}\}} (t-1){:}\mathit{ReplaceBulb} = \texttt{f} & \\
\quad \wedge\ (t-1){:}\mathit{Fault} = e \rightarrow t{:}\mathit{Choice}(\mathit{Fault} = e) & \mathcal{D}^{o\mathcal{BC}+}_{switch,6} \\
\bigwedge_{b \in \{\texttt{t},\texttt{f}\}} t{:}\mathit{ExtFault} = b \rightarrow t{:}\mathit{Fault} = b & \mathcal{D}^{o\mathcal{BC}+}_{switch,9} \\
(t-1){:}\mathit{ReplaceBulb} = \texttt{t} \rightarrow t{:}\mathit{Choice}(\mathit{Fault} = \texttt{u}) & \mathcal{D}^{o\mathcal{BC}+}_{switch,11} \\
t{:}\mathit{Choice}(\mathit{Fault} = \texttt{u}) & \sigma_{EF} \\
t{:}\mathit{UEC}(\mathit{Switch}) \wedge t{:}\mathit{UEC}(\mathit{Light}) \wedge t{:}\mathit{UEC}(\mathit{Fault}) & \mathit{UEC}(\sigma_F) \\
(t-1){:}\mathit{UEC}(\mathit{Flip}) \wedge (t-1){:}\mathit{UEC}(\mathit{ReplaceBulb}) & \mathit{UEC}(\sigma_A)
\end{array} \right.
$$

$$
Q[t] = \quad \neg\neg(0{:}\mathit{Switch} = \texttt{off} \wedge t{:}\mathit{Light} = \texttt{on}) \qquad \mathcal{Q}[t].
$$

Additionally,

- $I(B) = \{0{:}\mathit{Fault} = \texttt{t}, 0{:}\mathit{Fault} = \texttt{f}\}$,
- $I(P[t/i]) = \{i{:}\mathit{Fault} = \texttt{t}, i{:}\mathit{Fault} = \texttt{f}\}$, *and*
- $I(Q[t/i]) = \bigcup_{0 \leq j \leq i} \{i{:}\mathit{Fault} = \texttt{t}, i{:}\mathit{Fault} = \texttt{f}\}$.

Let $\mathbb{R}_{0,1} = \langle F, I, O \rangle$. *There is 1 stable model of* F*:*

$$A_1 = \left\{ \begin{array}{lll} 0{:}\mathit{Switch} = \mathtt{off}, & 0{:}\mathit{Light} = \mathtt{off}, & 0{:}\mathit{Fault} = \mathtt{u}, \\ 0{:}\mathit{ExtFault} = \mathtt{u}, & 0{:}\mathit{Flip} = \mathtt{t}, & 0{:}\mathit{ReplaceBulb} = \mathtt{f}, \\ 1{:}\mathit{Switch} = \mathtt{on}, & 1{:}\mathit{Light} = \mathtt{on}, & 1{:}\mathit{Fault} = \mathtt{u}, \\ 1{:}\mathit{ExtFault} = \mathtt{u} & & \end{array} \right\}.$$

Which corresponds to the histories $\mathcal{H}_{1,1}$ in Example 13.

Given $\mathcal{O}_{1,1}$ from Example 13, we have that

$$E_1[1] = \bigwedge \begin{array}{l} \neg\neg 0{:}\mathit{Flip} = \mathtt{t} \\ 1{:}\mathit{ExtFault} = \mathtt{t} \end{array} \qquad\qquad F_1[1] = \top.$$

There are then no stable models of $\mathbb{R}_{1,1}$ and 1 of $\mathbb{R}_{1,2}$:

$$A_2 = \left\{ \begin{array}{lll} 0{:}\mathit{Switch} = \mathtt{off}, & 0{:}\mathit{Light} = \mathtt{off}, & 0{:}\mathit{Fault} = \mathtt{u}, \\ 0{:}\mathit{ExtFault} = \mathtt{u}, & 0{:}\mathit{Flip} = \mathtt{t}, & 0{:}\mathit{ReplaceBulb} = \mathtt{f}, \\ 1{:}\mathit{Switch} = \mathtt{on}, & 1{:}\mathit{Light} = \mathtt{off}, & 1{:}\mathit{Fault} = \mathtt{t}, \\ 1{:}\mathit{ExtFault} = \mathtt{t}, & 1{:}\mathit{Flip} = \mathtt{f}, & 1{:}\mathit{ReplaceBulb} = \mathtt{t}, \\ 2{:}\mathit{Switch} = \mathtt{on}, & 2{:}\mathit{Light} = \mathtt{on}, & 2{:}\mathit{Fault} = \mathtt{u}, \\ 2{:}\mathit{ExtFault} = \mathtt{u} & & \end{array} \right\}.$$

This corresponds precisely to the history $\mathcal{H}_2$ in Example 13.

Additionally, we may equivalently express the online ASP encoding by prepending $\neg\neg$ to each external atom occurring in B and $P[t/i](i \geq 1)$ as well as each occurrence of an atom $(i{-}1){:}f = v$ where f is a fluent within $P[t/i]$. Given this alternate encoding, it holds that the resulting incremental theory and online progression is acyclic.

Proposition 11 (Acyclic Construction) *Given an Online $\mathcal{BC}+$ action description $\mathcal{D}$, observation stream $\mathcal{O}_{n,\tilde{m}}$, and some incrementally parametrized multi-valued formula $\mathcal{Q}[t]$, the alternate encodings of the incremental theory $\langle B, P[t], Q[t]\rangle^{\mathcal{D},\mathcal{Q}[t]}$ and online progression $\langle E, F\rangle_{\geq 1}^{\mathcal{O}_{n,\tilde{m}}}$ are acyclic.*

Due to this, as observed in section 6.1.1, we may reduce $\mathcal{BC}+$ into a disjunctive logic incremental theory and online progression to be computed by iClingo and oClingo.

Chapter 7

SYSTEM CPLUS2ASP

CPLUS2ASP was originally introduced by Casolary and Lee (2011) as a prototypical alternative to CCALC 2 (Giunchiglia *et al.* (2004)) for solving the action language $\mathcal{C}+$. While CCALC 2 compiled $\mathcal{C}+$ into propositional logic and used one of several available propositional SAT solvers as the reasoning workhorse, CPLUS2ASP uses the reduction provided in Section 3.4.3 in order to compile $\mathcal{C}+$ into propositional ASP and takes advantageous of the available highly-optimized ASP grounders and solvers. By doing this, Casolary and Lee observed an order of magnitude increase in performance over CCALC.

However, Casolary and Lee's CPLUS2ASP system was a proof-of-concept system, and, as such, was far from complete.

In this section we present a re-engineering of the CPLUS2ASP system, which we call CPLUS2ASP 2.0[1], that provides a number of improvements over the original system. Among these improvements are:

- an enhanced multi-modal architecture which allows for easily extending the system;
- support for multi-valued formulas under the Stable Mode Semantics and the action languages $\mathcal{BC}$ and $\mathcal{BC}+$, in addition to the existing support for $\mathcal{C}+$;
- syntactic support for external atom definitions via LUA scripting calls;
- an interactive command-line mode to allow users to easily setup program parameters;

[1] CPLUS2ASP 2 is available at `http://reasoning.eas.asu.edu/cplus2asp`.

- enhanced developer support via additional syntactic and static semantic checking;
- the ability to leverage the system iCLINGO for performing iterative deepening searches on the maximum length history being considered using a special case of the online $\mathcal{BC}+$ semantics provided in Section 6.4; and
- a new reactive running mode which allows for execution monitoring using an Online $\mathcal{BC}+$ description.

In addition to drastically improved useability, CPLUS2ASP 2.0's optimized standard library and ability to take advantage of incremental ASP computation techniques provides a significant increase in performance over its predecessors, CCALC and CPLUS2ASP v1, as well as the system COALA, which supports the evaluation of $\mathcal{B}$ and $\mathcal{C}$ via a reduction to ASP.

7.1 The Architecture of CPLUS2ASP

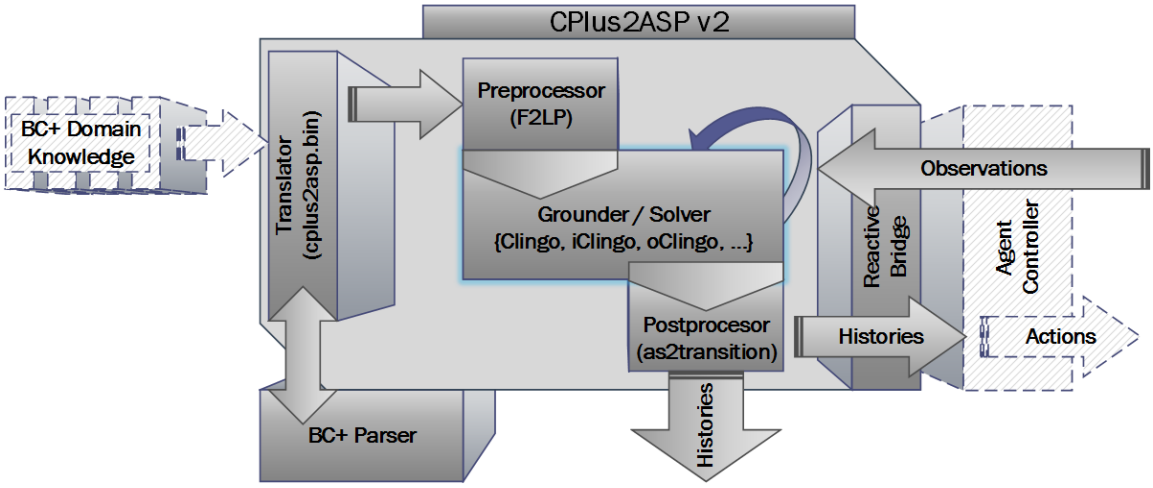

Figure 7.1: CPLUS2ASP v2 System Architecture

System CPLUS2ASP v2 is a re-engineering of the prototypical CPLUS2ASP v1 system (Casolary and Lee (2011)) and is available under version 3 of the GNU Public License.

Like its predecessor, CPLUS2ASP v2 uses a highly modular architecture that is designed to take advantage of the existing tools, including system F2LP and highly-optimized ASP grounders and solvers in addition to a number of packaged sub-components. A high-level conceptualization of the interaction of the sub-components in the CPLUS2ASP v2 system architecture can be seen in Figure 7.1.

The components within CPLUS2ASP are orchestrated by a wrapper application, also known as CPLUS2ASP. The wrapper is run in two modes: command-line and interactive. In both cases, the wrapper accepts configuration information from the user including, but certainly not limited to, the input language mode, the input files containing the problem description to examine, and the system running mode. The wrapper then uses this information to run each of the individual components, as follows:

Translator

First, the user provided problem description is passed to the translator application, CPLUS2ASP.BIN, which reduces one of several input languages into propositional formulas under the SM semantics using the appropriate reduction described in Chapter 3, Chapter 4, or Chapter 6.

Pre-processor

The wrapper then passes the resulting formulas to F2LP along with additional rules provided by an optimized standard library. F2LP uses the reduction process described by Lee and Palla in order to obtain an ASP program compatible with the target grounder/solver system.

Grounder/Solver

The program produced by F2LP is passed to the the grounder/solver system appropriate to the running mode. By default, this is

- CLINGO when running in *static mode*,
- ICLINGO when running in *incremental mode* (default), and
- OCLINGO when running in *reactive mode*.

The grounder/solver then produces the solutions to the translated problem description.

Post-processor

After a result has been returned by the grounder/solver system, the post-processor is used to produce a CCALC style transition system history from the solution.

In the event the system is running in static or incremental mode, the result is then returned to the user via the command line interface.

Reactive Bridge

The reactive bridge acts as an intermediary between the reactive solving system OCLINGO and a user-provided agent controller system. It allows the agent controller system to provide a $\mathcal{BC}+$ observation stream during execution and receive updated solutions in the form of transition system histories.

7.2 Modes of CPLUS2ASP

CPLUS2ASP is a multi-modal system which can be configured to behave in a number of fundamentally different ways. When discussing the modes of CPLUS2ASP we consider three different types of modes that can be configured:

input language mode

The *input language mode* determines which input language is currently being parsed by CPLUS2ASP. The input languages supported by CPLUS2ASP include multi-valued formulas under the Stable Model semantics and the action languages $\mathcal{C}+$, $\mathcal{BC}$, and $\mathcal{BC}+$.

interaction mode

The *interaction mode* determines what method is being used to gather configuration information from the user. While using the *command-line* interaction mode, the system gathers its configuration information exclusively from the command-line arguments provided by the user when calling CPLUS2ASP. Meanwhile, the *interactive* mode provides an interactive shell that the user is able to use to set basic parameters prior to running CPLUS2ASP.

running mode

The *running mode* determines the behavior of the system during the solving process as well as the target grounder/solver system.

The running modes, input language modes, and interaction modes are described in more detail in Sections 7.3, 7.4, and 7.5, respectively.

7.3 Running Modes of CPLUS2ASP

The running modes supported by CPLUS2ASP are *static-manual* mode, *static-auto* mode, *incremental* mode, and *reactive mode*. These are described in more depth as follows:

- In *static-manual* mode, the system targets the static ASP solver CLINGO and searches for histories of a fixed length, which may be specified by the user and modified after each run.

- In *static-auto* mode, the behavior is similar except that the length of the history is automatically incremented between the specified the minimum plan length (`minstep`) and maximum plan length (`maxstep`) until a solution is found.
- In *incremental* mode, the system targets the incremental ASP solver iClingo which is used to incrementally ground and solve the program while searching for a solution history of length k such that $\texttt{minstep} \leq k \leq \texttt{maxstep}$.
- Finally, in *reactive* mode, the system behaves similar to the incremental mode, except that it targets the reactive ASP solver oClingo which interfaces with a user application via the reactive bridge. This mode is only supported with the input language $\mathcal{BC}+$.

7.4 The Input Languages of Cplus2ASP

Cplus2ASP 2 is able to evaluate a number of different input languages via reduction into an Answer Set Program. Currently these are:

- multi-valued (propositional) formulas under SM (MVPF).
- the action language $\mathcal{C}+$,
- the action language $\mathcal{BC}$, and
- the online action language $\mathcal{BC}+$.

In this section, we discuss the input syntax for each of the languages in turn. We do this primarily by providing simplified fragments of the context-free grammar used to generate the translator's Lemon parser. A complete grammar as well as examples are included with Cplus2ASP's distribution files.

7.4.1 Shared Syntax

A Cplus2ASP program is a list of various statements. Each statement is divided into one of two basic categories: *declarations* and *laws*. A *declaration* provides meta

information for CPLUS2ASP, such as macro declarations, where to include other files, identifier types (sorts, constants, variables, objects), information on which constants should be shown in the program's output, and named query definitions. Meanwhile, a *law* is a rule in the program that is translated and evaluated.

Regardless of the input language, the set of available declarations are identical (although some options may very). In this section, we will discuss each of these available declarations and their impact on program evaluation. During the course of this, we will also provide a description of the base syntax of formulas within each of the laws, which will be used in the next sections while describing each of the available input languages.

Comments

CPLUS2ASP supports comments within programs which are stripped and passed through to the translated program via a pre-processor. Comments may begin with % or // and last until the end of the line. Alternatively, a comment beginning with /* lasts until a matching */ is encountered.

Placing comments within the translated program is a best effort task, and, as such, comments are not guaranteed to be exactly where they were placed originally. For example, a comment embedded within a statement will appear before the translated statement.

Macro Declarations and Expansion

CPLUS2ASP has, embedded within it, a fully functional macro expansion mechanism which is resolved during a single-pass two-stage parsing process. Macros are specified by one or more macro definition statements (`stmt_macro_def`) as is shown below.

```
stmt_macro_def ← :− macros macro_def_lst.
 macro_def_lst ← [macro_def_lst;] macro_bnd
     macro_bnd ← IDENTIFIER["("macro_args")"] −> MACRO_STRING
    macro_args ← [macro_args,]macro_arg
     macro_arg ← #INTEGER|#IDENTIFIER
```

Each macro is defined with a (possibly empty) set of macro expansion arguments, each beginning with a #, and an expansion string, which is anything after the $->$ and before a semicolon or period.

Once a macro has been defined, it can be used in any context. When used, each of the original macro expansion arguments occurring within the definition string are replaced with corresponding values provided to the macro and the result replaces the macro occurrence. The replacement process is performed iteratively from longest expansion argument to shortest.

Example 15 *For example, given the definition*

```
:- macros
        DEF(#1,#2) -> :- macros #1 -> #2;
        STRING(#x,#x2) -> "#x#x2bar".
```

The line

```
DEF(FOO,STRING(foo,bar)).
```

is expanded in two passes as follows:

```
DEF(FOO,STRING(foo,bar)).
        → :- macros FOO -> STRING(foo,bar).
        → :- macros FOO -> "foobarbar".
```

This results in an additional macro being defined named "`FOO`*."*

Include Statements

Include statements (`incl_stmt`) are very similar to macro statements except that they are expanded into the contents read in from one or more files. For each item in an include statement, the parser will attempt to find a file by that name either in the present working directory or the directory of the current file being operated on (in that order). The include statement is then replaced with concatenated contents of each of the files in the order they are listed.

```
incl_stmt ← :− include incl_lst.
 incl_lst ← [incl_lst,] incl_item
incl_item ← STRING_LITERAL|IDENTIFIER|INTEGER
```

Example 16 *The statement*

```
:- include one, ''two.cp'', 3.
```

will attempt to read from the files `one`*,* `two.cp`*, and* `3` *and replace the include statement with their contents concatenated together in that order.*

Sort Declarations

A *sort* is a named set of elements which is used in CPLUS2ASP to specify the domain of each constant and variable. Sorts are defined in two steps: the sort is first declared using a sort declaration statement (`stmt_sort_decl`) and, later, is defined by adding objects to it in an object declaration statement[2].

In addition to their individual contents, a sorts `s` can also automatically include the objects within another sort `s2` by defining `s` as supersort of `s1`, denoted `s` >> `s1` (or, equivalently, `s1` << `s`), during their declaration.

[2] Object declaration statements are introduced in Section 7.4.1.

```
stmt_sort_decl ← :− sorts sort_bnd_lst.
  sort_bnd_lst ← [sort_bnd_lst;]sort_bnd
      sort_bnd ← sort_dcl_lst
      sort_bnd ← sort_bnd << sort_bnd
      sort_bnd ← sort_bnd >> sort_bnd
      sort_bnd ← "("sort_bnd")"
  sort_dcl_lst ← [sort_dcl_lst,] IDENTIFIER
```

As a special note, CPLUS2ASP implicitly defines the `boolean` sort to range over the values `true` and `false`.

Example 17 *As an example, the statement*

```
:- sorts b; s >> (s1, s2).
```

defines 4 sorts: `b`*,* `s`*,* `s1`*, and* `s2`*. In addition, it also states that* `s` *should contain all of the elements within* `s1` *and* `s2`*.*

Object and Variable Declarations

In CPLUS2ASP an object is a value in a sort which a constant can take. It is also used in parameter lists to construct nested objects and sets of constants. Meanwhile, a variable is a placeholder symbol which will be replaced with each object in its domain during grounding and are invaluable for specifying any non-trivially sized problem.

Objects are specified using an object declaration statement (`stmt_object_def`) by providing three pieces of information:

- the base name of the object,
- a (possibly empty) list of sorts which act as ranges that each object parameter may take, and

- a sort which the object belongs to.

Variable declarations (`stmt_variable_def`) are specified in a similar manner, except variables cannot be declared with parameters.

```
stmt_object_def ← :− objects object_bnd_lst.
 object_bnd_lst ← [object_bnd_lst;] object_bnd
     object_bnd ← object_lst :: sort
     object_lst ← [object_lst,]object_spec
    object_spec ← IDENTIFIER["("sort_lst")"]
    object_spec ← NUMBER_RANGE
       sort_lst ← [sort_lst,]sort

stmt_variable_def ← :− variables variable_bnd_lst.
 variable_bnd_lst ← [variable_bnd_lst;]variable_bnd
     variable_bnd ← variable_lst :: sort
     variable_lst ← [variable_lst,]IDENTIFIER
```

As a shortcut, values in a integral numeric range from `n` to `m` can be declared succinctly using an expression of the form `n..m` (`NUMBER_RANGE`). A similar notation can be used for the object's parameters.

Example 18 *Assuming that the sorts* `int` *and* `s` *have been previously declared, the statement*

```
:- objects
        1..3            :: int;
        o(int, int)     :: s.
```

declares `1`,`2`, *and* `3` *as object within* `int` *and objects*

$$\mathtt{o}(1,1), \mathtt{o}(1,2), \mathtt{o}(1,3),$$

$$\mathtt{o}(2,1), \mathtt{o}(2,2), \mathtt{o}(2,3),$$

$$\mathtt{o}(3,1), \mathtt{o}(3,2), \textit{and } \mathtt{o}(3,3)$$

as values within `s`.

Furthermore, the declaration

```
:- variables
        I1, I2 :: int;
        S1, S2 :: s.
```

declares schematic variables `I1` *and* `I2` *to range over the objects within* `int`*, and variables* `S1` *and* `S2` *to range over the objects within* `s`.

Constant Declarations

As described in Chapters 3 and 4, constant symbols are the basic components of multi-valued formulas. Similar to object symbols, CPLUS2ASP constants are defined within a constant declaration statement (`stmt_const_def`) and have a base identifier, an optional list of parameter sort, and a sort which makes up the constant's domain. In addition, constants may have a type specifier (`constant_dcl_type`) which determines which group within the signature the constant belongs to.

```
stmt_const_def ← :− constants const_bnd_lst.
 const_bnd_lst ← [const_bnd_lst;] const_bnd
     const_bnd ← const_dcl_lst :: const_dcl_type["("sort")"]          (7.1)
     const_bnd ← const_dcl_lst :: sort                                (7.2)
     const_bnd ← const_dcl_lst :: attribute["("sort")"] of constant   (7.3)
 const_dcl_lst ← [const_dcl_lst,] IDENTIFIER["("sort_lst")"]
```

In (7.1), the value of `constant_dcl_type` is one of

```
abAction            action              additiveAction
additiveFluent      externalAction      externalAction
exogenousAction     inertialFluent      rigid
simpleFluent        sdFluent.
```

This corresponds to the constant's type specifier. (The valid values and meanings of each value vary depending on the input language.) Meanwhile, the value of `sort` provides the domain of the the constants in the list. `sort` can either be

- `sortname`, which specifies a predeclared sort,
- `n..m`, which specifies an implicit sort $\{i \mid n \leq i \leq m\}$, or
- `sortname`* or `sortname`$^\wedge$, are considered shorthand notations for specifying an unnamed sort consisting of the elements in `sortname` and the special element `none` or `unknown`, respectively.

(7.2) is shorthand for (7.1) such that `constant_dcl_type` is `rigid`. (7.3) declares an attribute to an existing action and is only valid in $\mathcal{BC}+$ and $\mathcal{C}+$.

Example 19 *Assuming that the sort* `int` *has been previously declared as* $\{1, 2\}$*, the statement*

```
:- constants
      p(int), q(int)  :: inertialFluent;
      a               :: exogenousAction(int*);
      b               :: attribute(1..5) of a.
```

declares Boolean fluents `p(1)`,`p(2)`,`q(1)`,`q(2)`*, the action* `a` *ranging over* `int` $\cup$ `{none}`*, and an action attribute* `b` *of* `a` *which ranges over* $\{1, 2, 3, 4, 5\}$.

Terms and Formulas

The most basic syntactic component of formulas and terms in CPLUS2ASP is the *base element* (`base_elem`), which essentially describes an instance of one of the symbols which have been previously declared or a LUA call, which will be described in the next section.

`base_elem` $\leftarrow$ `IDENTIFIER`[`(``term_lst``)`] `base_elem` $\leftarrow$ `lua`[3]
`term_lst` $\leftarrow$ [`term_lst`,]`term`

Each base element is classified according to its identifier and the number of parameters (arity) and matched to a previously declared symbol[4]. For example, a `constant` is a base element which matches a previously declared constant symbol.

A term (`term`) is built from any predeclared (non-sort) symbol, an integral value, andthe built-in Boolean values *true* and *false*. Numeric[5] terms may be combine with the standard unary arithmetic operators $-$ (negative) and *abs* (absolute value), and binary operators $+$ (addition), $-$ (subtraction), $*$ (multiplication), / (division), and *mod* (modulus).

`term` $:=$ `base_elem`	`term` $:=$ `INTEGER`	`term` $:=$ `STRING_LITERAL`
`term` $:=$ (`term`)	`term` $:=$ *true*	`term` $:=$ *false*
`term` $:=$ *maxstep*	`term` $:=$ *maxAdditive*	`term` $:=$ *maxAFValue*
`term` $:=$ $-$`term`	`term` $:=$ *abs* `term`	`term` $:=$ `term` $-$ `term`
`term` $:=$ `term` $+$ `term`	`term` $:=$ `term` $*$ `term`	`term` $:=$ `term`/`term`
`term` $:=$ `term` *mod* `term`		

Terms can be combined into a formulas in two essential ways: as an atomic formula (`af`) or as a comparison between two terms (`comparison`). Similar to previous sections, an atomic formula is an expression $c = v$ where c is a constant symbol and v is term. When c is Boolean, CPLUS2ASP allows the shorthand notations c and $\sim c$ which stand for $c =$ *true* and $c =$ *false*, respectively.

`af` $:=$ `constant`[$=$ `term`] `af` $:=$ $\sim$`constant`

[3] LUA functions will be reviewed in Section 7.4.1.

[4] Sorts and macros are exempt from this matching process as sorts cannot appear in formulas and macros are expanded prior to evaluation by the parser.

[5] A numeric term is an integer or a variable or constant with a domain consisting of integers, or an arithmetic combination of the two.

Comparison between two terms is quite straightforward. CPLUS2ASP supports the operators = (equality, synonymously ==), \= (inequality), < (less than), > (greater than), <= (less than or equal), and >= (greater than or equal). However, to disambiguate these from atomic formulas, we use a secondary term definition `term_strong` which essentially enforces that a constant cannot occur unless directly followed by an arithmetic or comparison operator (except =).

```
comparison :≔ term_strong = term
comparison :≔ term_strong \= term
comparison :≔ term_strong > term
comparison :≔ term_strong >= term
comparison :≔ -constant\ = term
comparison :≔ constant\ = term
comparison :≔ constant > term
comparison :≔ constant >= term
comparison :≔ term_strong == term
comparison :≔ term_strong < term
comparison :≔ term_strong =< term
comparison :≔ constant == term
comparison :≔ constant < term
comparison :≔ constant < term
comparison :≔ constant =< term
```

CPLUS2ASP supports fully nested formulas constructed from atomic formulas and term comparisons as well as two convenient formula abbreviations: quantifier formulas (big conjunction/disjunction)[6] (`formula_quant`) and cardinality formulas (`formula_card`). These can be combined with the standard unary logical operator *not* (synonymously, −) as well as the binary operators & (conjunction), | (disjunction), −> (implication), and <−> (equivalence).

[6] As the universe of a CPLUS2ASP program is fixed and finite the quantifiers ∀ and ∃ may be equivalently represented as a big conjunction or big disjunction (respectively) over the names of elements in the universe. Due to this, we use these two notions interchangeably.

```
formula_base ::= comparison          formula_base ::= af
formula_base ::= formula_quant       formula_base ::= formula_card
formula_base ::= true                formula_base ::= false
     formula ::= formula_base             formula ::= (formula)
     formula ::= not formula              formula ::= -formula
     formula ::= formula & formula        formula ::= formula"|"formula
     formula ::= formula -> formula       formula ::= formula <-> formula
```

Quantifier formulas and cardinality formulas are as follows:

```
formula_quant ::= "["quant_lst"|"formula"]"
    quant_lst ::= [quant_lst] quant_op variable
     quant_op ::= /\
     quant_op ::= \/

 formula_card ::= [term_strong]{variable_lst"|"formula}[term]
 variable_lst ::= [variable_lst,] variable
```

Essentially, a quantifier formula is an abbreviation of a conjunction (or disjunction) over all the elements in the domains of the variables on the left-hand side.

For example, assuming that $V1$ and $V2$ are variables ranging over $\{1, 2\}$, the quantifier

$$[\backslash/V1/\backslash V2 \mid \texttt{foo}(V1, V2)]$$

is shorthand for

$$(\texttt{foo}(1, 1)\ \&\ \texttt{foo}(1, 2)) \mid (\texttt{foo}(2, 1)\ \&\ \texttt{foo}(2, 2)).$$

Intuitively, a cardinality formula $n\{\mathbf{x}|F(\mathbf{x})\}$ states that there are at least n possible assignments to the variables within $\mathbf{x}$ such that $F(\mathbf{x})$ is true. While $\{\mathbf{x}|F(\mathbf{x})\}n$ is

similar, except there are at most n possible assignments of $\mathbf{x}$. Finally, $n\{\mathbf{x}|F(\mathbf{x})\}m$ intuitively states that there are between n and m such assignments.

Formally, these are shorthand for nested formulas. $n\{\mathbf{x}|F(\mathbf{x})\}$ can be expanded to

$$\bigwedge_{1\leq i\leq n} F(\mathbf{x}_i) \wedge \bigwedge_{1\leq i\leq j\leq n} \neg(\mathbf{x}_i = \mathbf{x}_j)$$

where each $\mathbf{x}_i$ is a distinct list of variables of the same length as $\mathbf{x}$ and, for any two lists $\mathbf{x} = x_1, \ldots, x_m$ and $\mathbf{y} = y_1, \ldots, y_m$, $\mathbf{x} = \mathbf{y}$ is shorthand for $x_1 = y_1 \wedge \cdots \wedge x_n = y_n$.

Additionally, $\{\mathbf{x}|F(\mathbf{x})\}n$ can be viewed as shorthand for $\neg(n+1\{\mathbf{x}|F(\mathbf{x})\})$ and $n\{\mathbf{x}|F(\mathbf{x})\}m$ can be seen as $n\{\mathbf{x}|F(\mathbf{x})\}$ & $\{\mathbf{x}|F(\mathbf{x})\}m$.

LUA

System CPLUS2ASP 2 allows for embedding external LUA function calls in the system, which are evaluated by the grounder at grounding time. These LUA calls allow the user a great deal of flexibility when designing a program and can be used for complex computation that is not easily expressible in logic programs. LUA functions may be defined within the scope of a LUA code block, which is began by the statement

```
:- begin lua.
```

and ended by the statement

```
:- end lua.
```

These code blocks are not parsed by CPLUS2ASP and, instead, are left to the ASP grounder/solver. As such, CPLUS2ASP does not verify the existence of a LUA function during translation.

Given a LUA function defined within a code block, it may then be called by prepending the "`@`" symbol to the function call.

$$\texttt{lua} ::= \texttt{@IDENTIFIER}[(\texttt{term\_lst})]$$

Example 20 *Given the LUA definition*

```
:- begin lua.
        function a(x)
                return x + 2
        end
:- end lua.
```

The formula

```
c= @a(1)
```

evaluates to

```
c= 3
```

Show/Hide Statements

Show and Hide statements allow the user to control which constants are displayed in the transition system histories output by CPLUS2ASP. Each statement may either specify a list of atomic formulas show (or hide) or the keyword *all*, which causes the statement to affect all constants.

$$\texttt{stmt\_show} ::= :- (\textit{show} \mid \textit{hide})\ \texttt{af\_lst}.$$
$$\texttt{stmt\_show} ::= :- (\textit{show} \mid \textit{hide})\ \textit{all}.$$

$$\texttt{af\_lst} ::= [\texttt{af\_conj}\ \&]\ \texttt{af}$$

(Note that each atomic formula $\texttt{c}(\mathbf{x}) = \texttt{v}$ occurring in a show or hide statement cannot contain any constants occurring within $\mathbf{x}$ or v.)

When consecutive statements conflict, the later one will override the previous ones.

Example 21 *Assuming* `X` *is a variable ranging over* $\{1, 2\}$*, the statements*

```
:- hide all.
:- show p(X), q(1)=2.
```

will hide all atomic formulas in the output histories except `p(1) = true`*,* `p(2) = true`*, and* `q(1) = 2`*.*

Queries

A *query* is a named set of constraints on the transition system history which is generated by CPLUS2ASP. Each query has three components: a label, an optional maximum step (or maximum step range), and a set of constraint formulas to apply, each parametrized with the step at which they should be applied (or `maxstep` to be applied at the current maximum step).

```
          stmt_query ::= :- query query_lst.
           query_lst ::= [query_lst;] term_no_const : formula
           query_lst ::= [query_lst;] query_maxstep_decl
           query_lst ::= [query_lst;] query_label_decl
query_maxstep_decl ::= maxstep :: INTEGER
query_maxstep_decl ::= maxstep :: NUMBER_RANGE
  query_label_decl ::= label :: INTEGER
  query_label_decl ::= label :: IDENTIFIER
```

(`term_no_const` is similar to `term`, except it contains no constant symbols.)

A query should have at most one label and maximum step definition. In the event no label is provided, the label is assumed to be "0"; meanwhile, in the event

no maximum step definition is provided the maximum step range is 0.. inf (i.e. the system will try longer and longer histories indefinitely until a solution is found).

Example 22 *As an example, the statement*

```
:- query
        label::foo;
        maxstep:: 0..10;
        0: p(1) & not p(2);
        maxstep: p(2).
```

specifies a query named `foo` *which searches for histories between lengths* `0` *and* `10` *such that the initial state satisfies* `p(1)` & *not* `p(2)` *and the final state satisfies* `p(2)`.

7.4.2 Multi-Valued (Propositional) Formulas (MVPF)

CPLUS2ASP provides support for programs consisting of multi-valued formulas under the Stable Model Semantics. When configured to run in this input mode CPLUS2ASP expects each constant to be declared as a `rigid` constant (or equivalently, without a type specifier) which indicates that each of these constant are time in-variant.

There is a single type of law in an multi-valued program (`law_mvpf`), which is an expression $F \leftarrow G$ standing for the implication $G \rightarrow F$. These laws, like all other laws supported by CPLUS2ASP also support an optional *where* clause, which is evaluated during grounding time to provide additional scope for each rule.

`law` ::= `formula_base`[<− `formula`] [`where`].

`where` ::= *where* `formula_no_const`

Two important things to note are that arbitrary formulas in the head of each rule is not currently supported. Instead, the head is allowed to be a single formula element,

```
:- macros N -> 8.

:- sorts num.

:- objects  1..N :: num.


:- variables I, I1, J, J1, NX :: num.

:- constants
  q(num,num) :: num.


{q(I,J)=N}.
<- q(I,J)=NX & q(I1,J)=NX & I\=I1.
<- q(I,J)=NX & q(I,J1)=NX & J\=J1.
<- q(I,J)=NX & q(I1,J1)=NX & I\=I1 & abs(I1-I)=J1-J.
```

Figure 7.2: 8-queens in MVPF

and that the *where* clause is evaluated completely during the grounding process and cannot contain constants (`formula_no_const` is a constant-free version `formula`).

The multi-valued formula input language is not current compatible with queries as well as the incremental and reactive running modes.

As an example, Figure 7.4.2 provides an encoding of the N-queens problem, where N queens must be placed on an $N \times N$ chess board such that no queen can attack another, in the input language of MVPF.

7.4.3 The Action Language $\mathcal{C}+$

The action language $\mathcal{C}+$ is CPLUS2ASP's namesake and the original action language supported by the CPLUS2ASP version 1. It was described in Section 3.4.3.

The valid constant types in $\mathcal{C}+$ are as follows:

action A basic action within the signature;

exogenousAction Shorthand for an action which is also exogenous (i.e. the law "**exogenous** a" is added for each such action a);

abAction Shorthand for an action which defaults to false (the law "**default** a = `false`" is added for each such action);

additiveAction An additive action constant, discussed later;

additiveFluent An additive fluent constant, discussed later;

simpleFluent A simple fluent in the signature;

inertialFluent A simple fluent which is inertial (the law "**inertial** f" is added for each such fluent);

sdFluent A statically determined fluent within the signature;

rigid A time-invariant rigid constant.

CPLUS2ASP supports each of the law shorthands discussed in Section 3.4.3 in addition to several additional shorthands. Each of these laws provide a number of optional clauses which have distinct meanings depending on the law they occur in. These can include any of those provided below as well as `where`, which, as in MVPF, provides ground-time evaluated formulas.

`if` ::= *if* `formula`
`after` ::= *after* `formula`
`unless` ::= *unless* `constant`
`by` ::= *by* `term`

An if clause (`if`) provides a formula matched against the current state of the system (as well as actions being executed when the clause occurs within a *causes* or *may cause* style law). Conversely, an after clause (`after`) provides a formula which is matched against the previous state and any actions that may have been performed. The unless clause (`unless`) specifies a, possibly undeclared, abnormality action that serves to

disable the law when asserted; when the abnormality is undeclared, the presence of the clause *unless* c is viewed as shorthand for declaring c as a Boolean `abAction` and adding the law

default `c=false`.

Finally, the by clause (`by`) is used for providing values to increment and decrement additive constants, which will be discussed in later in this section.

`law` :≔ *caused* `cplus_head` [`if`] [`after`] [`unless`] [`where`]. (7.4)
`law` :≔ *possibly caused* `cplus_head` [`if`] [`after`] [`unless`] [`where`]. (7.5)
`law` :≔ `formula` *causes* `cplus_head` [`if`] [`unless`] [`where`]. (7.6)
`law` :≔ `formula` *may cause* `cplus_head` [`if`] [`unless`] [`where`]. (7.7)
`law` :≔ *always* `formula` [`after`] [`unless`] [`where`]. (7.8)
`law` :≔ *constraint* `formula` [`after`] [`unless`] [`where`]. (7.9)
`law` :≔ *impossible* `formula` [`after`] [`unless`] [`where`]. (7.10)
`law` :≔ *never* `formula` [`after`] [`unless`] [`where`]. (7.11)
`law` :≔ *default* `af` [`if`] [`after`] [`unless`] [`where`]. (7.12)
`law` :≔ *exogenous* `constant` [`if`] [`after`] [`unless`] [`where`]. (7.13)
`law` :≔ *inertial* `constant` [`if`] [`after`] [`unless`] [`where`]. (7.14)
`law` :≔ *nonexecutable* `formula` [`if`] [`unless`] [`where`]. (7.15)
`law` :≔ *rigid* `constant` [`where`]. (7.16)
`law` :≔ `constant` *increments* `constant by` [`if`] [`unless`] [`where`]. (7.17)
`law` :≔ `constant` *decrements* `constant by` [`if`] [`unless`] [`where`]. (7.18)
(7.19)

`cplus_head` :≔ `af`
`cplus_head` :≔ *true*
`cplus_head` :≔ *false*

In addition to the shorthand laws provided in Section 3.4.3, we provide:

- the defeasible causal law (7.5) such that

possibly caused F **after** $H_1 \wedge H_2$;

is synonymous with for

$$\textbf{default } F \textbf{ after } H_1 \wedge H_2;$$

- the defeasible dynamic formed law (7.7) such that

$$H_1 \textbf{ may cause } F \textbf{ if } H_2$$

stands for

$$\textbf{possibly caused } F \textbf{ after } H_1 \wedge H_2;$$

- the constraint law (7.8) which is synonymous with (7.9);
- the negative constraint laws (7.10) and (7.11), such that

$$\textbf{impossible } G \textbf{ after } H, \text{ and}$$

$$\textbf{never } G \textbf{ after } H$$

are shorthand for

$$\textbf{constraint } \neg G \textbf{ after } H;$$

- the constraint law (7.16) such that

$$\textbf{rigid } c$$

is shorthand for the set of constraints

$$\textbf{constraint } c = v \textbf{ after } c = v \qquad\qquad v \in \mathit{Dom}(c); \text{ and}$$

- the additive laws (7.17) and (7.18), which will be discussed momentarily.

As an example, Figure 7.4.3 shows the pendulum problem discussed earlier described in the input language of $\mathcal{C}+$.

```
:- constants
  right       :: simpleFluent;
  hold        :: exogenousAction.

hold causes right if right.
hold causes -right if -right.

default right after -right.
default -right after right.

:- query
maxstep :: 2..2;
0: -right.
```

Figure 7.3: The Pendulum Problem in $\mathcal{C}+$

Additive Constants

Lee and Lifschitz (2003) describe a method of encoding *additive constants* within $\mathcal{C}+$ in order to provide an elaboration tolerant way of modeling the effect of multiple contributions onto a single constant.

CPLUS2ASP supports additive action and fluents constants, which cannot occur in the head of any law which is not a *increments* (7.17) or *decrements* (7.18) law. Their behavior is as follows:

- If the constant c is an additive fluent begin with its previous value, otherwise begin with 0.
- Add v to the value of c for each *increments* law

  $$a \textbf{ increments } c \textbf{ by } v \textbf{ if } G$$

  such that $G \wedge a = \texttt{true}$ was satisfied in the previous state and transition.
- Similarly, subtract v from the value of c for each *decrements* law

  $$a \textbf{ decrements } c \textbf{ by } v \textbf{ if } G$$

  such that $G \wedge a = \texttt{true}$ was satisfied in the previous state and transition.

Finally, in order to provide support for these constants, CPLUS2ASP, like its predecessors, needs an upper bound on the values which all of the contributions can amount to. This is provided by setting the value for `maxAdditive` using a declaration

$$\texttt{stmt\_maxadditive} ::= :- \mathit{maxAdditive} = \texttt{INTEGER}.$$

or by defining it using the command line interface at run time.

7.4.4 The Action Language $\mathcal{BC}$

The action language $\mathcal{BC}$ was discussed in Section 3.4.4 and is also supported by CPLUS2ASP[7]. In $\mathcal{BC}$, the valid constant types are

action A basic action within the signature (assumed to be exogenous);

simpleFluent A simple fluent in the signature;

inertialFluent A simple fluent which is inertial (the law "**inertial** f" is added for each such fluent);

sdFluent A statically determined fluent within the signature;

rigid A time-invariant rigid constant.

The laws supported by $\mathcal{BC}$ are similar to $\mathcal{C}+$ with several notable exceptions:

- action constants cannot occur in the heads of any law,
- the bodies of each law must be a conjunction of atoms,
- many of the laws have an additional ifcons clause, as discussed in Section 3.4.4,
- additive constants and the *increments* and *decrements* laws are not supported.

[7] CPLUS2ASP actually supports a slight extension to $\mathcal{BC}$ in which it is possible to utilize multi-valued actions.

`ifcons` ← *ifcons* `formula`

`law` :≔ `cplus_head` [`if`] [`ifcons`] [`after`] [`unless`] [`where`]. (7.20)
`law` :≔ `formula` *causes* `cplus_head` [`if`] [`unless`] [`where`]. (7.21)
`law` :≔ *always* `formula` [`after`] [`unless`] [`where`]. (7.22)
`law` :≔ *constraint* `formula` [`after`] [`unless`] [`where`]. (7.23)
`law` :≔ *impossible* `formula` [`after`] [`unless`] [`where`]. (7.24)
`law` :≔ *never* `formula` [`after`] [`unless`] [`where`]. (7.25)
`law` :≔ *default* `af` [`if`] [`ifcons`] [`after`] [`unless`] [`where`]. (7.26)
`law` :≔ *inertial* `constant` [`if`] [`ifcons`] [`after`] [`unless`] [`where`]. (7.27)
`law` :≔ *nonexecutable* `formula` [`if`] [`unless`] [`where`]. (7.28)
`law` :≔ *rigid* `constant` [`where`]. (7.29)

7.4.5 The Action Language $\mathcal{BC}+$

CPLUS2ASP also supports the online and offline variants of the action language $\mathcal{BC}$ as described in Sections 4.1 and 6.2.

While running in $\mathcal{BC}+$ mode, the system allows for all previously discussed constant types in addition to two new types: `externalAction` and `externalFluent` which are externally defined constants.

The laws available in $\mathcal{BC}+$ are similar syntactically to those available in $\mathcal{C}+$, with the notable exception that any base formula element, including cardinality formulas and quantifiers, may now be included in the head of each law.

`law` :≔ `formula_base` [`if`] [`after`] [`unless`] [`where`]. (7.30)
`law` :≔ *possibly caused* `formula_base` [`if`] [`after`] [`unless`] [`where`]. (7.31)
`law` :≔ `formula` *causes* `formula_base` [`if`] [`unless`] [`where`]. (7.32)
`law` :≔ `formula` *may cause* `formula_base` [`if`] [`unless`] [`where`]. (7.33)
`law` :≔ *always* `formula` [`after`] [`unless`] [`where`]. (7.34)
`law` :≔ *constraint* `formula` [`after`] [`unless`] [`where`]. (7.35)
`law` :≔ *impossible* `formula` [`after`] [`unless`] [`where`]. (7.36)
`law` :≔ *never* `formula` [`after`] [`unless`] [`where`]. (7.37)
`law` :≔ *default* `af` [`if`] [`after`] [`unless`] [`where`]. (7.38)
`law` :≔ *exogenous* `constant` [`if`] [`after`] [`unless`] [`where`]. (7.39)
`law` :≔ *inertial* `constant` [`if`] [`after`] [`unless`] [`where`]. (7.40)
`law` :≔ *nonexecutable* `formula` [`if`] [`unless`] [`where`]. (7.41)
`law` :≔ *rigid* `constant` [`where`]. (7.42)
`law` :≔ `constant` *increments* `constant` `by` [`if`] [`unless`] [`where`]. (7.43)
`law` :≔ `constant` *decrements* `constant` `by` [`if`] [`unless`] [`where`]. (7.44)
(7.45)

In the event that a cardinality formula occurs within the head of a law, it should have the form

$$n\{\mathbf{x}|\ F(\mathbf{X})\}m$$

such that F is a single atomic formula and $\mathbf{x}$ is a list of schematic variables $x_1, \ldots, x_k$. This occurrence is considered shorthand for the the choice formula

$$\bigwedge_{e_1 \in Dom(x_1)} \wedge \cdots \wedge \bigwedge_{e_k \in Dom(x_k)} (F(\mathbf{e}) \vee \neg F(\mathbf{e}))$$

such that $\mathbf{e}$ is $e_1, \ldots, e_k$ and the rule

$$\textbf{constraint}\ n\{\mathbf{x}|\ c(t_1(\mathbf{x}), \ldots, t_l(\mathbf{x})) = v(\mathbf{x})\}m.$$

While running in the reactive mode, system CPLUS2ASP solves for plans which are normal with respect to the provided input stream. This means that external constants are assumed to be `unknown` until it is asserted in the online progression.

7.5 Running Cplus2ASP

After crafting an input program for Cplus2ASP in one of the supported languages, the next logical step is, of course, to run the program using Cplus2ASP.

Cplus2ASP v2 currently offers two distinct user-interaction methods: command-line and interactive shell. The command-line mode is designed primarily for interacting with an automated system, such as a script or user provided application, or a seasoned Cplus2ASP user who is familiar with the various options available in Cplus2ASP. Meanwhile, the interactive mode provides users with a friendlier interface for providing the various settings which Cplus2ASP requires to run with minimal effort from the user.

7.5.1 Using the Command-Line Mode

The command-line running mode is a non-interactive mode which accepts settings from the program's command line arguments and provides a single result returned from the solving process. The Cplus2ASP system defaults to this running mode when all required information, such as the query to run and values for special constants such as `maxAdditive` are provided (when required).

A Cplus2ASP command-line call take the form of

```
cplus2asp <FILES> [<OPTIONS>] [<CONSTANTS>] [<SOLUTIONS>].
```

Each of these input sections are described below in some detail:

<**FILES**> A list of input files including the provided input program and any supporting files relative to the present working directory. Certain extensions, such as `.fof` and `.lp` are handled with special care. For instance, these extensions will bypass the translator and be passed directly to the pre-processor and grounder / solver, respectively.

<**OPTIONS**> A number of optional input options, such as the target input language and instructions to skip or replace certain tool chain components. These are discussed further later.

<**CONSTANTS**> A list of command-line definitions for simple macros that the user may provide in order to affect the behavior of his or her program. Each of these definitions are of the form

```
<IDENTIFIER>=<VALUE>
```

and is equivalent to prepending the macro definition

```
:- macros <IDENTIFIER> -> <VALUE>.
```

to the beginning of the program.

In practice, these provide a convenient way to specify program parameters at run time without modifying the program file. This allows the user to try many variations of these parameters rapidly.

In the event < `IDENTIFIER` > is either "`query`", "`maxAdditive`", "`minstep`", or "`maxstep`", these are treated as specifications for these special non-macro values.

<**SOLUTIONS**> The number of solution histories that should be found and outputted. If this number is "`0`" or the keyword "`all`" then all solutions are generated and returned. If no number is provided, the system defaults to finding a single solution.

Command-Line Options

CPLUS2ASP is highly-configurable and, as such, offers a variety of command-line options in order to specify various behaviors of the program, including the input

language and running mode of the application. In this section, we briefly introduce the basic command line options. Which involve selecting the system's input language, running mode, the query to be executed, and minimum/maximum step information for that query.

When executing a program, CPLUS2ASP needs to know what input language the program has been written in in order to properly interpret it. This is done by using the

```
--language=<LANG>
```

option, where $<$ `LANG` $>$ is "`MVPF`", "`C+`", "`BC`", or "`BC+`". By default, CPLUS2ASP assumes the program is composed in the action language $\mathcal{C}+$.

The system's default running mode is dependent on the input language selected. Typically, the system runs in incremental mode and targets ICLINGO as its solver. However, the system defaults to the static-auto mode for MVPF programs and reactive mode for $\mathcal{BC}+$ program which contain external constants. The user may also override the system's default running mode by providing the

```
--mode=<MODE>
```

option, where $<$ `MODE` $>$ is "`static − auto`", "`static − manual`", "`incremental`", or "`reactive`".[8]

Finally, unless the system's input language is MVPF, it is necessary to specify a query to run. This is done using the

`--query=<QUERY>`

[8] It is important to note that there are certain restrictions for input language / running mode combinations. MVPF programs must run in the static-auto mode. Meanwhile, reactive mode is valid iff the input program is a $\mathcal{BC}+$ program with external constants.

option, where < QUERY > is one of the query labels defined within the input program, or a built in label such as `sat`, `states`, and `transitions`. The `sat` label specifies that the program should be checked for satisfiability and no query should be enforced, the `states` label refers to a query find valid states of the transition system, likewise, the `transitions` label finds valid transitions in the transition system.

The user may also override the query-defined maximum step range. This is done with the options

```
--minstep=<MIN>, and
--maxstep=<MAX>
```

where < MIN > and < MAX > are the minimum and maximum lengths of histories to attempt while testing the query. In the event only a maxstep is specified, it is a assumed that the minimum and maximum steps are the same (i.e. the system should only test one history length). Alternatively, the user may provide

```
--maxstep=<MIN>..<MAX>
```

to specify the minimum and maximum steps together.

7.5.2 Using the Interactive Mode

The interactive mode provides a shell-like interface which allows the user to perform many of the configurations available from the command line. In general, the user-interactive mode is entered any time the user fails to provide all necessary information within the command-line arguments. As such, the easiest way to enter the user-interactive mode is to neglect to specify a query on the command-line.

Regardless of whether the user wishes to use the interactive mode or command-line mode, at a minimum CPLUS2ASP requires that the user pre-specify the files

that should be parsed as well as the input language the program is written in (if it is different from the default).

As an example, the command

```
cplus2asp switch.cp --language=c+
```

will parse the program within `switch.cp` according to the syntax and semantics of $\mathcal{C}+$ and then enter user-interactive mode.

While in the user-interactive mode, the following commands, among others, are available to the user:

`help` Displays the list of available commands.
`config` Reveals the currently selected running options.
`queries` Displays the list of available queries to run.
`minstep=[#]` Overrides the minimum step to solve for for the next query selected.
`maxstep=[#]` Overrides the maximum step to solve for for the next query selected.
`sol=[#]` Selects the number of solutions to display.
`query=[QUERY]` Runs the selected query and returns the results.
`exit` Exits the program.

If the user wished to find all the states in the program, he could then type the commands

```
sol=all
query=states.
```

Following successful execution of a query, the system will return to the interactive prompt and the process can be repeated. Afterward, if he wished to find all transitions he could simply type

```
query=transitions.
```

Options such as `minstep`, `maxstep`, and `sol` also accept the keyword `default` to return the option to its default setting as selected by the query being ran, the command line options, and the system defaults.

For more information on using CPLUS2ASP v2, we invite the reader to explore the documentation available at `http://reasoning.eas.asu.edu/cplus2asp` or within the help usage message available by executing `cplus2asp --help`.

7.6 Experiments

In this section, we present the results of benchmarking CPLUS2ASP 2's offline running modes against its predecessors as well as a similar system COALA, which serve as a demonstration of the effectiveness of CPLUS2ASP's various running modes.

In order to compare the performance of the CPLUS2ASP 2 system with its predecessors, we used large variants of several widely known domains [9] and compared the performance of CPLUS2ASP's offline running modes with the performance of CCALC v2, CPLUS2ASP v1, and the incremental and static running modes of COALA (where applicable). All experiments were performed on an Intel Core 2 Duo 3.00 GHZ CPU with 4 GB RAM running Ubuntu 11.10. The CCALC v2 tests used RELSAT 2.0 as a SAT solver while CPLUS2ASP v1, v2, and COALA tests used the same version of CLINGO, v3.0.5.

The domains tested include large variants of the Traffic World (Akman *et al.* (2004)), a numerical domain which models the behavior of cars on a road; a variant of the Blocks World, as described in Lee and Lifschitz (2003); the Spacecraft Integer (Lee and Lifschitz (2003)), which models a spacecraft's movement with multiple inde-

[9] All benchmark programs are available at
http://reasoning.eas.asu.edu/cplus2asp/benchmarks.

Domain	#	CCalc 2	Cplus2ASP v1	coala		Cplus2ASP v2	
				static	incr.	static	incr.
traffic (altmerge)	11	878.59s + 1s [a] [531552 / 3671940]	95.43s + 25.95s [2722247 / 3341068]	– [b] –	– –	82.16s + 26.57s [2262231 / 2766459]	14.2s + 2.6s
bw-cost (15) [c]	8	131.1s + 5s [149032 / 624439]	76.16s + 0.4s [123517 / 260282]	–	–	17.09s + 3.16s [43052 / 526923]	3.47s + 0.16s
bw-cost (20)	9	52s + 987s [374785 / 1584778]	271s + 9.17s [279869 / 626496]	–	–	63.26s + 66.58s [102426 / 1745166]	13.45s + 2.24s
spacecraft (15/8) [d]	3	173.62s + 0s [128262 / 622158]	16.07s + 2.65s [146056 / 146056]	–	–	5.57s + 0.06s [132918 / 253514]	2.33s + 0.01s
spacecraft (25/10)	4	*timeout*	208.2s + 480.24s [760673 / 1653650]	–	–	67.55s + 3.42s [732860 / 1427771]	17.46s + 0.35s
hanoi (6/3) [e]	64	14s + 1983s [13710 / 221895]	38.9s + 137.27s [37297 / 298047]	1039.15s + 507.12s [13798 / 410559]	1.4s + 51.13s	547.9s + 47.53s [10086 / 202694]	0.76s + 3.5s
towers (8/4)	33	*timeout*	31.19s + 102.69s [35041 / 433660]	304.02s + 3017.87s [12922 / 655436]	1.51s + 470.23s	102.81s + 89.36s [9074 / 324668]	1.04s + 14.8s
ferryman (10/4) [f]	16	39.45s + 0s [55905 / 308909]	8.27s + 2.98s [14122 / 120693]	40.85s + 8.71s [4973 / 358772]	0.87s + 1.85s	21.59s + 2.37s [12721 / 112912]	0.66s + 0.25s
ferryman (15/4)	26	1004.26s + 0s [5] [256590 / 1452554]	85.21s + 39.54s [42687 / 539513]	793.13s + 169.18s [15718 / 2275992]	6.13s + 14.73s	318.4s + 34.4s [39536 / 515167]	4.18s + 2.97s

Figure 7.4: Benchmarking Results

[a] preprocessing time + solve time [# atoms / # rules]
[b] '–' means that the input language of coala is not expressive enough to represent the domain.
[c] maximum cost
[d] domain size (15 × 15 × 15) / goal position
[e] disks / pegs
[f] # animals / boat capacity

pendent jets; the Towers of Hanoi; and the Ferryman domain, which involves moving a number of wolves and sheep across a river without allowing the sheep to be eaten.

The Towers of Hanoi and Ferryman descriptions are from examples packaged with coala v1.0.1. In order to run them on other systems, we manually converted them CCalc's input syntax.

Table 7.4 compares the results of the test benchmarks for each of the available configurations. Each measured time includes translation, grounding, and solving for all possible maximum steps between 0 and the horizon (#), as well as the number of atoms and rules produced below each timing. In all test cases Cplus2ASP's incre-

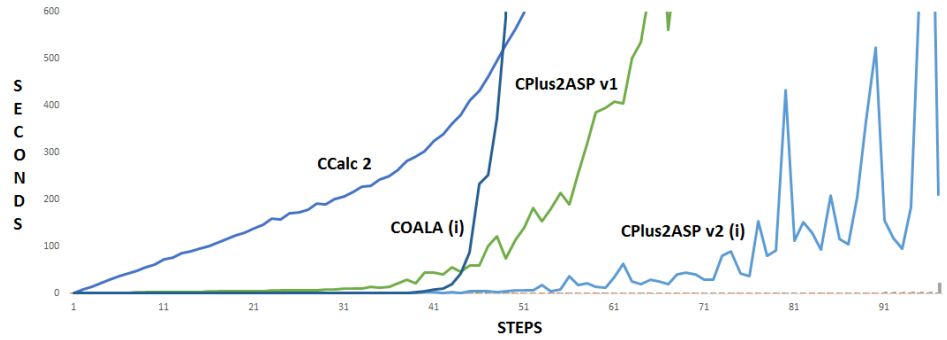

Figure 7.5: Ferryman 120/4 Long Horizon Analysis

mental running mode showed a significant performance advantage compared to the other systems, performing roughly 3 times faster than COALA's incremental mode and an order of magnitude faster than its predecessor CPLUS2ASP v1. COALA's incremental running mode comes in the second place in all but one benchmark. CPLUS2ASP v2's static mode tended to outperform its predecessor on the more computation-heavy additive domains, but was subsequently outmatched in the others. Finally, CCALC 2 and COALA's static mode came in last (with CCALC performing slightly worse in most cases).

Figure 7.5 shows a more detailed analysis of the execution of the first 100 steps of solving an extreme variant of the ferryman domain consisting of 120 of each animal by graphing the time spent (in seconds) on each step by each configuration. While the static configurations were required to completely re-ground and re-solve the translated answer set program for each maximum step, resulting in an ever-growing amount of work to be performed at each step, CPLUS2ASP v2's incremental running mode is able to avoid this by only grounding the new cumulative ($P[t]$) and volatile ($Q[t]$) components and leveraging heuristics learned from previous iterations. This results in far less time being required for checking each increment.

Although COALA's incremental mode uses the same reasoning engine, ICLINGO, as CPLUS2ASP v2's incremental mode, system CPLUS2ASP sees a significant overall

speed-up over COALA, which can be attributed to a significant reduction in the number of rules produced during grounding due to the availability of multi-valued fluents in CPLUS2ASP resulting in a more succinct action description as well as a translation optimized for the heuristics applied by ICLINGO, which results in far fewer conflicts and restarts during solving in all test cases.

Chapter 8

CONCLUSION

Modeling dynamic systems is an important Knowledge Representation problem which has received attention due to its use as a reasoning platform for the evolution of real-world environments. We examined and compared a number of existing techniques for modeling these systems, including Answer Set Programming and the action languages $\mathcal{B}$, $\mathcal{C}$, $\mathcal{C}+$, and $\mathcal{BC}$. While ASP provides an expressive semantics capable of modeling many interesting problems, the low-level logical syntax and lack of structure makes non-trivial development in ASP difficult. Meanwhile, the action languages each provide a highly structured syntax and intuitive semantics for modeling dynamic systems, but this comes at the price of stifling losses in expressivity.

To make matters worse, none of these techniques are capable of efficiently performing an iterative deepening search, such as searching for the shortest plan to accomplish a goal, or gracefully handle exceptions which occur while executing such a plan. In either case, the traditional answer is to repeatedly restart the grounding/-solving process from scratch, throwing out any current results. In practice this proves quite limiting as the intractable nature of solving each of these formalism makes each of these restarts prohibitively expensive.

In this work we provided a promising solution to these deficiencies in the form of an integrated framework for incremental and online reasoning. Our framework consists of a new online action language, which we call $\mathcal{BC}+$, whose offline fragment provides a proper generalization of each of the aforementioned action languages while maintaining their high-level structure and transition system semantics. We showed how this action language can then be solved in an incremental fashion, allowing

for efficiently performing an iterative deepening search to generate minimum length histories, using an online ASP theory based on the background theory of the ASP system oCLINGO (and, as a special case, iCLINGO). In addition, $\mathcal{BC}+$ also provides facilities for efficiently handling exceptions in an online environment using the same theories.

Finally, we provided an implementation of this framework in the form of the system Cplus2ASP 2.0, which implements a number of the formalisms discussed including multi-valued propositional formulas under the Stable Model Semantics and the action languages $\mathcal{C}+$, $\mathcal{BC}$, and $\mathcal{BC}+$. In practice, CPLUS2ASP shows a performance increase of roughly an order of magnitude when compared to its predecessor as well as providing additional enhancements in functionality and useability.

Although the online $\mathcal{BC}+$ framework shows promise for solving several of the deficiencies present within existing formalisms, its effectiveness has yet to be fully evaluated. As such, future work will focus on exploring specific applications for $\mathcal{BC}+$, such as a robot agent controller, and fully evaluating its applicability to these applications. In addition, additional extensions of the framework should be considered based on the recent extension of the Stable Model Semantics to allow for generalized quantifiers (and, as a special case, ASP aggregates).

This work has been published in part within

- Babb and Lee (2012), "Module theorem for the general theory of stable models", Theory and Practice of Logic Programming **12**, 4-5, 719–735 (2012), and
- Babb and Lee (2013), "Cplus2asp: Computing action language $\mathcal{C}+$ in answer set programming", in "Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)", pp. 122–134 (2013).

5

REFERENCES

Akman, V., S. Erdoğan, J. Lee, V. Lifschitz and H. Turner, "Representing the Zoo World and the Traffic World in the language of the Causal Calculator", Artificial Intelligence **153(1–2)**, 105–140 (2004).

Babb, J. and J. Lee, "Module theorem for the general theory of stable models", TPLP **12**, 4-5, 719–735 (2012).

Babb, J. and J. Lee, "Cplus2asp: Computing action language $\mathcal{C}+$ in answer set programming", in "Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)", pp. 122–134 (2013).

Bartholomew, M. and J. Lee, "Stable models of formulas with intensional functions", in "Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)", pp. 2–12 (2012).

Cabalar, P. and P. Ferraris, "Propositional theories are strongly equivalent to logic programs", TPLP **7**, 6, 745–759 (2007).

Casolary, M. and J. Lee, "Representing the language of the Causal Calculator in Answer Set Programming", in "Technical Communications of the 27th International Conference on Logic Programming (ICLP)", pp. 51–61 (2011).

Erdem, E., K. Haspalamutgil, C. Palaz, V. Patoglu and T. Uras, "Combining high-level causal reasoning with low-level geometric reasoning and motion planning for robotic manipulation", in "ICRA", pp. 4575–4581 (2011).

Ferraris, P., "Answer sets for propositional theories", in "Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)", pp. 119–131 (2005).

Ferraris, P., "A logic program characterization of causal theories", in "Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)", pp. 366–371 (2007).

Ferraris, P., J. Lee, Y. Lierler, V. Lifschitz and F. Yang, "Representing first-order causal theories by logic programs", Theory and Practice of Logic Programming **12**, 3, 383–412 (2012).

Ferraris, P., J. Lee and V. Lifschitz, "Stable models and circumscription", Artificial Intelligence **175**, 236–263 (2011).

Ferraris, P., J. Lee, V. Lifschitz and R. Palla, "Symmetric splitting in the general theory of stable models", in "Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)", pp. 797–803 (2009a).

Ferraris, P., J. Lee, V. Lifschitz and R. Palla, "Symmetric splitting in the general theory of stable models", in "Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)", pp. 797–803 (2009b).

Gebser, M., T. Grote, R. Kaminski, P. Obermeier, O. Sabuncu and T. Schaub, "Stream reasoning with answer set programming: Preliminary report", in "Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning (KR'12)", edited by T. Eiter and S. McIlraith, pp. 613–617 (AAAI Press, 2012).

Gebser, M., T. Grote, R. Kaminski and T. Schaub, "Reactive answer set programming", in "Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)", pp. 54–66 (2011a).

Gebser, M., T. Grote and T. Schaub, "Coala: a compiler from action languages to ASP", in "Proceedings of European Conference on Logics in Artificial Intelligence (JELIA)", pp. 169–181 (2010).

Gebser, M., R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub and S. Thiele, "Engineering an incremental ASP solver", in "Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08)", edited by M. Garcia de la Banda and E. Pontelli, vol. 5366 of *Lecture Notes in Computer Science*, pp. 190–205 (Springer-Verlag, 2008).

Gebser, M., R. Kaminski, A. König and T. Schaub, "Advances in *gringo* series 3", in "Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)", edited by J. Delgrande and W. Faber, vol. 6645 of *Lecture Notes in Artificial Intelligence*, pp. 345–351 (Springer-Verlag, 2011b).

Gebser, M., R. Kaminski, M. Ostrowski, T. Schaub and S. Thiele, "On the input language of ASP grounder *Gringo*", in "Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)", edited by E. Erdem, F. Lin and T. Schaub, vol. 5753 of *Lecture Notes in Artificial Intelligence*, pp. 502–508 (Springer-Verlag, 2009).

Gebser, M., B. Kaufmann, A. Neumann and T. Schaub, "CLASP: A conflict-driven answer set solver", in "Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)", (2007a).

Gebser, M., T. Schaub and S. Thiele, "Gringo: A new grounder for answer set programming", in "Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning", pp. 266–271 (2007b).

Gelfond, M. and V. Lifschitz, "The stable model semantics for logic programming", in "Proceedings of International Logic Programming Conference and Symposium", edited by R. Kowalski and K. Bowen, pp. 1070–1080 (MIT Press, 1988).

Gelfond, M. and V. Lifschitz, "Classical negation in logic programs and disjunctive databases", New Generation Computing **9**, 365–385 (1991).

Gelfond, M. and V. Lifschitz, "Action languages [1] ", Electronic Transactions on Artificial Intelligence **3**, 195–210 (1998).

Gelfond, M. and V. Lifschitz, "The common core of action languages $\mathcal{B}$ and $\mathcal{C}$ [2] ", in "Working Notes of the International Workshop on Nonmonotonic Reasoning (NMR)", (2012).

Giunchiglia, E., J. Lee, V. Lifschitz, N. McCain and H. Turner, "Nonmonotonic causal theories", Artificial Intelligence **153(1–2)**, 49–104 (2004).

Giunchiglia, E. and V. Lifschitz, "An action language based on causal explanation: Preliminary report", in "Proceedings of National Conference on Artificial Intelligence (AAAI)", pp. 623–630 (AAAI Press, 1998).

Janhunen, T., E. Oikarinen, H. Tompits and S. Woltran, "Modularity aspects of disjunctive stable models", in "Procedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)", pp. 175–187 (2007).

Janhunen, T., E. Oikarinen, H. Tompits and S. Woltran, "Modularity aspects of disjunctive stable models", Journal of Artificial Intelligence Research **35**, 813–857 (2009).

Lee, J., *Automated Reasoning about Actions* [3] , Ph.D. thesis, University of Texas at Austin (2005).

Lee, J., "Reformulating action language $\{C\}$ + in answer set programming", in "Correct Reasoning", pp. 405–421 (2012).

Lee, J. and V. Lifschitz, "Describing additive fluents in action language $\mathcal{C}+$", in "Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)", pp. 1079–1084 (2003).

Lee, J., V. Lifschitz and F. Yang, "Action language BC: Preliminary report", in "Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)", (2013).

Lee, J. and Y. Meng, "Stable models of formulas with generalized quantifiers", in "14th International Workshop on Non-Monotonic Reasoning (NMR 2012)", (2012).

Lee, J. and R. Palla, "Yet another proof of the strong equivalence between propositional theories and logic programs", in "Working Notes of the Workshop on Correspondence and Equivalence for Nonmonotonic Theories", (2007).

Lee, J. and R. Palla, "System F2LP — computing answer sets of first-order formulas", in "Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)", pp. 515–521 (2009).

[1] `http://www.ep.liu.se/ea/cis/1998/016/`
[2] `http://www.cs.utexas.edu/users/vl/papers/bc.pdf`
[3] `http://peace.eas.asu.edu/joolee/papers/dissertation.pdf`

Leone, N. and et al., "A disjunctive datalog system dlv (2005-02-23)", in "University of Calabria, Vienna University of Technology", (2005), available under `http://www.dbai.tuwien.ac.at/proj/dlv/`.

Leone, N., W. Faber, G. Pfeifer, T. Eiter, G. Gottlob, S. Perri and F. Scarcello, "The DLV system for knowledge representation and reasoning", ACM Transactions on Computational Logic **7**, 3, 499–562 (2006a).

Leone, N., G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri and F. Scarcello, "The dlv system for knowledge representation and reasoning", ACM Trans. Comput. Log. **7**, 3, 499–562 (2006b).

Lifschitz, V. and H. Turner, "Splitting a logic program", in "Proceedings of International Conference on Logic Programming (ICLP)", edited by P. Van Hentenryck, pp. 23–37 (1994).

Lifschitz, V. and H. Turner, "Representing transition systems by logic programs", in "Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)", pp. 92–106 (1999).

Lifschitz, V. and F. Yang, "Translating first-order causal theories into answer set programming", in "Proceedings of the European Conference on Logics in Artificial Intelligence (JELIA)", pp. 247–259 (2010).

McCain, N., *Causality in Commonsense Reasoning about Actions* [4] , Ph.D. thesis, University of Texas at Austin (1997).

Oikarinen, E. and T. Janhunen, "Modular equivalence for normal logic programs", in "17th European Conference on Artificial Intelligence(ECAI)", pp. 412–416 (2006).

Son, T. C., C. Baral, T. H. Nam and S. A. McIlraith, "Domain-dependent knowledge in answer set planning", CoRR **cs.AI/0207023** (2002).

[4] `ftp://ftp.cs.utexas.edu/pub/techreports/tr97-25.ps.gz`

APPENDIX A

REFERENCED ACTION DESCRIPTIONS

Named Sets:	Value:		
Status	$\{\texttt{on}, \texttt{off}\}$		
Boolean	$\{\texttt{t}, \texttt{f}\}$		
Constants:	**Type:**		**Domain:**
Sw, *Light*	Fluent		*Status*
Flip	Action		*Boolean*
Flip **causes** $Sw = \texttt{on}$ **if** $Sw = \texttt{off}$.			$\mathcal{D}^{\mathcal{B}}_{\text{switch},1}$
Flip **causes** $Sw = \texttt{off}$ **if** $Sw = \texttt{on}$.			$\mathcal{D}^{\mathcal{B}}_{\text{switch},2}$
Light $= s$ **if** $Sw = s$.		$s \in$ *Status*	$\mathcal{D}^{\mathcal{B}}_{\text{switch},3}$

Figure A.1: The Light Switch Problem in $\mathcal{B}$

Named Sets:	Value:		
Status	$\{\texttt{on}, \texttt{off}\}$		
Switch	$\{\texttt{s1}, \texttt{s2}, \dots\}$		
Boolean	$\{\texttt{f}, \texttt{f}\}$		
Constants:	**Type:**		**Domain:**
$Sw(x)$	Fluent	$x \in$ *Switch*	*Status*
$Flip(x)$	Action	$x \in$ *Switch*	*Action*
$Flip(x)$ **causes** $Sw(x) = \texttt{on}$ **if** $Sw(x) = \texttt{off}$.		$x \in$ *Switch*	$\mathcal{D}^{\mathcal{B}}_{\text{switch2},1}$
$Flip(x)$ **causes** $Sw(x) = \texttt{off}$ **if** $Sw(x) = \texttt{on}$.		$x \in$ *Switch*	$\mathcal{D}^{\mathcal{B}}_{\text{switch2},2}$
$Sw(x) = \texttt{off}$ **if** $Sw(y) = \texttt{on}, Sw(\texttt{s3}) = \texttt{on}$.		$x, y \in \{\texttt{s1}, \texttt{s2}\}, x \neq y$	$\mathcal{D}^{\mathcal{B}}_{\text{switch2},3}$
$Sw(x) = \texttt{on}$ **if** $Sw(y) = \texttt{off}, Sw(\texttt{s3}) = \texttt{on}$.		$x, y \in \{\texttt{s1}, \texttt{s2}\}, x \neq y$	$\mathcal{D}^{\mathcal{B}}_{\text{switch2},4}$

Figure A.2: The Many Switch Problem in $\mathcal{B}$

Named Sets:	Value:		
Status	$\{\texttt{on}, \texttt{off}\}$		
Boolean	$\{\texttt{t}, \texttt{f}\}$		
Constants:	Type:		Domain:
Sw, *Light*	Fluent		*Status*
Flip	Action		*Boolean*
inertial *Sw*.			$\mathcal{D}^{\mathcal{C}}_{\text{switch},1}$
caused $Sw = \texttt{on}$ **after** $Flip = \texttt{t} \wedge Sw = \texttt{off}$.			$\mathcal{D}^{\mathcal{C}}_{\text{switch},2}$
caused $Sw = \texttt{off}$ **after** $Flip = \texttt{t} \wedge Sw = \texttt{on}$.			$\mathcal{D}^{\mathcal{C}}_{\text{switch},3}$
caused $Light = s$ **if** $Sw = s$.		$s \in Status$	$\mathcal{D}^{\mathcal{C}}_{\text{switch},4}$

Figure A.3: The Light Switch Problem in $\mathcal{C}$

Named Sets:	Value:		
Location	$\{\texttt{left}, \texttt{right}\}$		
Boolean	$\{\texttt{t}, \texttt{f}\}$		
Constants:	Type:		Domain:
Arm	Fluent		*Location*
Hold	Action		*Boolean*
caused $Arm = \texttt{left}$ **if** $Arm = \texttt{left}$ **after** $Arm = \texttt{right}$.			$\mathcal{D}^{\mathcal{C}}_{\text{pendulum},1}$
caused $Arm = \texttt{right}$ **if** $Arm = \texttt{right}$ **after** $Arm = \texttt{left}$.			$\mathcal{D}^{\mathcal{C}}_{\text{pendulum},2}$
caused $Arm = l$ **after** $Hold = \texttt{t} \wedge Arm = l$.		$l \in Location$	$\mathcal{D}^{\mathcal{C}}_{\text{pendulum},3}$

Figure A.4: The Pendulum Problem in $\mathcal{C}$

A.3 The Action Language $\mathcal{C}+$

Named Sets:	Value:		
Status	$\{\texttt{on}, \texttt{off}\}$		
Boolean	$\{\texttt{t}, \texttt{f}\}$		
Constants:	**Type:**		**Domain:**
Sw, *Light*	Simple Fluent		*Status*
Flip	Action		*Boolean*
inertial *Sw*.			$\mathcal{D}^{\mathcal{C}+}_{\text{switch},1}$
exogenous *Flip*.			$\mathcal{D}^{\mathcal{C}+}_{\text{switch},2}$
Flip = t **causes** *Sw* = on **if** *Sw* = off.			$\mathcal{D}^{\mathcal{C}+}_{\text{switch},3}$
Flip = t **causes** *Sw* = off **if** *Sw* = on.			$\mathcal{D}^{\mathcal{C}+}_{\text{switch},4}$
caused *Light* = *s* **if** *Sw* = *s*.		$s \in \mathit{Status}$	$\mathcal{D}^{\mathcal{C}+}_{\text{switch},5}$

Figure A.5: The Light Switch Problem in $\mathcal{C}+$

Named Sets:	Value:		
Location	$\{\texttt{left}, \texttt{right}\}$		
Boolean	$\{\texttt{t}, \texttt{f}\}$		
Constants:	**Type:**		**Domain:**
Arm	Fluent		*Location*
Hold	Action		*Boolean*
exogenous *Hold*.			$\mathcal{D}^{\mathcal{C}+}_{\text{pendulum},1}$
default *Arm* = left **after** *Arm* = right.			$\mathcal{D}^{\mathcal{C}+}_{\text{pendulum},2}$
default *Arm* = right **after** *Arm* = left.			$\mathcal{D}^{\mathcal{C}+}_{\text{pendulum},3}$
Hold = t **causes** *Arm* = *l* **if** *Arm* = *l*.		$l \in \mathit{Location}$	$\mathcal{D}^{\mathcal{C}+}_{\text{pendulum},4}$

Figure A.6: The Pendulum Problem in $\mathcal{C}+$

Named Sets:	Value:
Len	$\{\texttt{1}, \texttt{2}, \ldots, \texttt{none}\}$
Type	$\{\texttt{journal}, \texttt{conference}, \texttt{workshop}, \texttt{none}\}$
Boolean	$\{\texttt{t}, \texttt{f}\}$

Constants:	Type:	Domain:
HasPub, *HasLongPub*	Fluent	*Boolean*
HasJournalPub	Fluent	*Boolean*
Publish	Action	*Boolean*
PubType	Action	*Type*
PubLen	Action	*Len*

inertial *HasPub*.		$\mathcal{D}^{\mathcal{C}+}_{\text{publish},1}$
inertial *HasLongPub*.		$\mathcal{D}^{\mathcal{C}+}_{\text{publish},2}$
inertial *HasJournalPub*.		$\mathcal{D}^{\mathcal{C}+}_{\text{publish},3}$
exogenous *Publish*.		$\mathcal{D}^{\mathcal{C}+}_{\text{publish},4}$
exogenous *PubType*.		$\mathcal{D}^{\mathcal{C}+}_{\text{publish},5}$
exogenous *PubLen*.		$\mathcal{D}^{\mathcal{C}+}_{\text{publish},6}$
constraint $\mathit{PubType} = \texttt{none} \leftrightarrow \mathit{Publish} = \texttt{f}$.		$\mathcal{D}^{\mathcal{C}}_{\text{publish},7}$
constraint $\mathit{PubLen} = \texttt{none} \leftrightarrow \mathit{Publish} = \texttt{f}$.		$\mathcal{D}^{\mathcal{C}}_{\text{publish},8}$
$\mathit{Publish} = \texttt{t}$ **causes** $\mathit{HasPub} = \texttt{t}$.		$\mathcal{D}^{\mathcal{C}}_{\text{publish},9}$
$\mathit{PubLen} = x$ **causes** $\mathit{HasLongPub} = \texttt{t}$.	$x \in \mathit{Len}, x > 30$	$\mathcal{D}^{\mathcal{C}}_{\text{publish},10}$
$\mathit{PubType} = \texttt{journal}$ **causes** $\mathit{HasJournalPub} = \texttt{t}$.		$\mathcal{D}^{\mathcal{C}}_{\text{publish},11}$
constraint $\mathit{HasLongPub} = \texttt{t} \rightarrow \mathit{HasPub} = \texttt{t}$.		$\mathcal{D}^{\mathcal{C}}_{\text{publish},12}$
constraint $\mathit{HasJournalPub} = \texttt{t} \rightarrow \mathit{HasPub} = \texttt{t}$.		$\mathcal{D}^{\mathcal{C}}_{\text{publish},13}$

Figure A.7: The Publishing Problem in $\mathcal{C}+$

Named Sets:	Value:		
Status	$\{\texttt{on}, \texttt{off}\}$		
Boolean	$\{\texttt{t}, \texttt{f}\}$		
Constants:	**Type:**		**Domain:**
Sw, *Light*	Fluent		*Status*
Flip	Action		*Boolean*
inertial Sw.			$\mathcal{D}^{\mathcal{BC}}_{\text{switch},1}$
Flip **causes** $Sw = \texttt{on}$ **if** $Sw = \texttt{off}$.			$\mathcal{D}^{\mathcal{BC}}_{\text{switch},2}$
Flip **causes** $Sw = \texttt{off}$ **if** $Sw = \texttt{on}$.			$\mathcal{D}^{\mathcal{BC}}_{\text{switch},3}$
$Light = s$ **if** $Sw = s$.		$s \in Status$	$\mathcal{D}^{\mathcal{BC}}_{\text{switch},4}$

Figure A.8: The Light Switch Problem in $\mathcal{BC}$

Named Sets:	Value:		
Status	$\{\texttt{on}, \texttt{off}\}$		
Switch	$\{\texttt{s1}, \texttt{s2}, \dots\}$		
Boolean	$\{\texttt{t}, \texttt{f}\}$		
Constants:	**Type:**		**Domain:**
$Sw(x)$	Fluent	$x \in Switch$	*Status*
$Flip(x)$	Action	$x \in Switch$	*Boolean*
inertial $Sw(x)$		$x \in Switch$	$\mathcal{D}^{\mathcal{BC}}_{\text{switch2},1}$
$Flip(x)$ **causes** $Sw(x) = \texttt{on}$ **if** $Sw(x) = \texttt{off}$.		$x \in Switch$	$\mathcal{D}^{\mathcal{BC}}_{\text{switch2},2}$
$Flip(x)$ **causes** $Sw(x) = \texttt{off}$ **if** $Sw(x) = \texttt{on}$.		$x \in Switch$	$\mathcal{D}^{\mathcal{BC}}_{\text{switch2},3}$
$Sw(x) = \texttt{off}$ **if** $Sw(y) = \texttt{on}, Sw(\texttt{s3}) = \texttt{on}$.		$x, y \in \{\texttt{s1}, \texttt{s2}\}, x \neq y$	$\mathcal{D}^{\mathcal{BC}}_{\text{switch2},4}$
$Sw(x) = \texttt{on}$ **if** $Sw(y) = \texttt{off}, Sw(\texttt{s3}) = \texttt{on}$.		$x, y \in \{\texttt{s1}, \texttt{s2}\}, x \neq y$	$\mathcal{D}^{\mathcal{BC}}_{\text{switch2},5}$

Figure A.9: The Many Switch Problem in $\mathcal{BC}$

Named Sets:	Value:		
Len	$\{\mathtt{1}, \mathtt{2}, \dots\}$		
Type	$\{\mathtt{journal}, \mathtt{conference}, \mathtt{workshop}\}$		
Boolean	$\{\mathtt{t}, \mathtt{f}\}$		
Action	$\{\mathtt{pub}(l,t), \mathtt{wait}\}$	$(l,t) \in \mathit{Len} \times \mathit{Type}$	

Constants:	Type:	Domain:
HasPub, *HasLongPub*	Fluent	*Boolean*
HasJournalPub	Fluent	*Boolean*
a	Action	*Action*

inertial *HasPub*.		$\mathcal{D}^{\mathcal{BC}}_{\text{publish},1}$
inertial *HasLongPub*.		$\mathcal{D}^{\mathcal{BC}}_{\text{publish},2}$
inertial *HasJournalPub*.		$\mathcal{D}^{\mathcal{BC}}_{\text{publish},3}$
$\mathtt{pub}(l,t)$ **causes** $\mathit{HasPub} = \mathtt{t}$.	$(l,t) \in \mathit{Len} \times \mathit{Type}$	$\mathcal{D}^{\mathcal{BC}}_{\text{publish},4}$
$\mathtt{pub}(l,t)$ **causes** $\mathit{HasLongPub} = \mathtt{t}$.	$(l,t) \in \mathit{Len} \times \mathit{Type}$ $l > 30$	$\mathcal{D}^{\mathcal{BC}}_{\text{publish},5}$
$\mathtt{pub}(l, \mathtt{journal})$ **causes** $\mathit{HasJournalPub} = \mathtt{t}$.	$l \in \mathit{Len}$	$\mathcal{D}^{\mathcal{BC}}_{\text{publish},6}$
$\mathit{HasPub} = \mathtt{t}$ **if** $\mathit{HasJournalPub} = \mathtt{t}$.		$\mathcal{D}^{\mathcal{BC}}_{\text{publish},7}$
$\mathit{HasPub} = \mathtt{t}$ **if** $\mathit{HasLongPub} = \mathtt{t}$.		$\mathcal{D}^{\mathcal{BC}}_{\text{publish},8}$

Figure A.10: The Publishing Problem in $\mathcal{BC}$

A.5 The Action Language $\mathcal{BC}+$

Named Sets:	Value:		
Status	$\{\texttt{on}, \texttt{off}\}$		
Switch	$\{\texttt{s1}, \texttt{s2}, \dots\}$		
Boolean	$\{\texttt{t}, \texttt{f}\}$		
Constants:	**Type:**		**Domain:**
$Sw(x)$	Fluent	$x \in Switch$	*Status*
$Flip(x)$	Action	$x \in Switch$	*Boolean*
inertial $Sw(x)$		$x \in Switch$	$\mathcal{D}^{\mathcal{BC}+}_{\text{switch2,1}}$
$Flip(x) = \texttt{t}$ **causes** $Sw(x) = \texttt{on}$ **if** $Sw(x) = \texttt{off}$.		$x \in Switch$	$\mathcal{D}^{\mathcal{BC}+}_{\text{switch2,2}}$
$Flip(x) = \texttt{t}$ **causes** $Sw(x) = \texttt{off}$ **if** $Sw(x) = \texttt{on}$.		$x \in Switch$	$\mathcal{D}^{\mathcal{BC}+}_{\text{switch2,3}}$
$Sw(x) = \texttt{off}$ **if** $Sw(y) = \texttt{on}, Sw(\texttt{s3}) = \texttt{on}$.		$x, y \in \{\texttt{s1}, \texttt{s2}\}, x \neq y$	$\mathcal{D}^{\mathcal{BC}+}_{\text{switch2,4}}$
$Sw(x) = \texttt{on}$ **if** $Sw(y) = \texttt{off}, Sw(\texttt{s3}) = \texttt{on}$.		$x, y \in \{\texttt{s1}, \texttt{s2}\}, x \neq y$	$\mathcal{D}^{\mathcal{BC}+}_{\text{switch2,5}}$

Figure A.11: The Many Switch Problem in $\mathcal{BC}+$

Named Sets:	Value:		
Location	$\{\texttt{left}, \texttt{right}\}$		
Boolean	$\{\texttt{t}, \texttt{f}\}$		
Constants:	**Type:**		**Domain:**
Arm	Fluent		*Location*
Hold	Action		*Boolean*
exogenous *Hold*.			$\mathcal{D}^{\mathcal{BC}+}_{\text{pendulum,1}}$
default $Arm = \texttt{left}$ **after** $Arm = \texttt{right}$.			$\mathcal{D}^{\mathcal{BC}+}_{\text{pendulum,2}}$
default $Arm = \texttt{right}$ **after** $Arm = \texttt{left}$.			$\mathcal{D}^{\mathcal{BC}+}_{\text{pendulum,3}}$
$Hold = \texttt{t}$ **causes** $Arm = l$ **if** $Arm = l$.		$l \in Location$	$\mathcal{D}^{\mathcal{BC}+}_{\text{pendulum,4}}$

Figure A.12: The Pendulum Problem in $\mathcal{BC}+$

Named Sets:	Value:
Len	$\{\texttt{1}, \texttt{2}, \ldots, \texttt{none}\}$
Type	$\{\texttt{journal}, \texttt{conference}, \texttt{workshop}, \texttt{none}\}$
Boolean	$\{\texttt{t}, \texttt{f}\}$

Constants:	Type:	Domain:
HasPub, *HasLongPub*	Fluent	*Boolean*
HasJournalPub	Fluent	*Boolean*
Publish	Action	*Boolean*
PubType	Action	*Type*
PubLen	Action	*Len*

inertial *HasPub*.		$\mathcal{D}^{\mathcal{BC}+}_{\text{publish},1}$
inertial *HasLongPub*.		$\mathcal{D}^{\mathcal{BC}+}_{\text{publish},2}$
inertial *HasJournalPub*.		$\mathcal{D}^{\mathcal{BC}+}_{\text{publish},3}$
exogenous *Publish*.		$\mathcal{D}^{\mathcal{BC}+}_{\text{publish},4}$
exogenous *PubType*.		$\mathcal{D}^{\mathcal{BC}+}_{\text{publish},5}$
exogenous *PubLen*.		$\mathcal{D}^{\mathcal{BC}+}_{\text{publish},6}$
constraint $\mathit{PubType} = \texttt{none} \leftrightarrow \mathit{Publish} = \texttt{f}$.		$\mathcal{D}^{\mathcal{BC}}_{\text{publish},7}$
constraint $\mathit{PubLen} = \texttt{none} \leftrightarrow \mathit{Publish} = \texttt{f}$.		$\mathcal{D}^{\mathcal{BC}}_{\text{publish},8}$
$\mathit{Publish} = \texttt{t}$ **causes** $\mathit{HasPub} = \texttt{t}$.		$\mathcal{D}^{\mathcal{BC}}_{\text{publish},9}$
$\mathit{PubLen} = x$ **causes** $\mathit{HasLongPub} = \texttt{t}$.	$x \in \mathit{Len}, x > 30$	$\mathcal{D}^{\mathcal{BC}}_{\text{publish},10}$
$\mathit{PubType} = \texttt{journal}$ **causes** $\mathit{HasJournalPub} = \texttt{t}$.		$\mathcal{D}^{\mathcal{BC}}_{\text{publish},11}$
constraint $\mathit{HasLongPub} = \texttt{t} \rightarrow \mathit{HasPub} = \texttt{t}$.		$\mathcal{D}^{\mathcal{BC}}_{\text{publish},12}$
constraint $\mathit{HasJournalPub} = \texttt{t} \rightarrow \mathit{HasPub} = \texttt{t}$.		$\mathcal{D}^{\mathcal{BC}}_{\text{publish},13}$

Figure A.13: The Publishing Problem in $\mathcal{BC}+$

APPENDIX B

PROOFS OF STATEMENTS

B.1 Proposition 2

Lemma 2 *Given formulas $F_1 = F \wedge H \wedge G$ and $F_2 = F \wedge H \wedge Choice(A)$ where A is a set of atoms such that F is negative on A and G is negative on $At(F) \setminus A$, it holds that if a set of atoms I is an answer set of F_1 it is an answer set of F_2.*

Proof.

As F is negative on A and both G and $Choice(A)$ is negative on $At(F) \setminus A$, it holds via Theorem 3 of Babb and Lee (2012) that

$$SM[F \wedge H \wedge G; \sigma] \leftrightarrow SM[F \wedge H; \sigma \setminus A] \wedge SM[H \wedge G; A], \text{ and}$$
$$SM[F \wedge H \wedge Choice(A); \sigma] \leftrightarrow SM[F \wedge H; \sigma \setminus A] \wedge SM[H \wedge Choice(A); A]$$

Furthermore, by Theorem 2 of Ferraris *et al.* (2011) it holds that

$$\begin{aligned} SM[F \wedge H; \sigma \setminus A] \wedge SM[H \wedge Choice(A); A] &\leftrightarrow SM[F \wedge H; \sigma \setminus A] \wedge SM[H; \emptyset] \\ &\leftrightarrow SM[F \wedge H; \sigma \setminus A] \wedge H \end{aligned}$$

Finally, by Lemma 1 of Ferraris *et al.* (2009b) it holds that

$$SM[F \wedge H; \sigma \setminus A] \rightarrow H$$

therefore

$$SM[F \wedge H; \sigma \setminus A] \wedge H \leftrightarrow SM[F \wedge H; \sigma \setminus A].$$

As

$$SM[F \wedge H; \sigma \setminus A] \wedge SM[H \wedge G; A] \rightarrow SM[F \wedge H; \sigma \setminus A]$$

it trivially follows that

$$SM[F \wedge H \wedge G; \sigma] \rightarrow SM[F \wedge H \wedge Choice(A); \sigma].$$

For convenience we refer to the notation presented in Babb and Lee (2012). As such, given a signature σ, set of atoms $I \subseteq \sigma$, and formula F such that $At(F) \subseteq \sigma$ we say that $I \subseteq \sigma$ is an answer set of a formula F iff $I \models SM[F; \sigma]$.

Proposition 2
Given a $\mathcal{BC}+$ action description $\mathcal{D}^{\mathcal{BC}+}$ and any $k \geq 0$ it holds that the histories of length k of $\mathcal{T}(\mathcal{D}^{\mathcal{BC}+})$ correspond exactly to the answer sets of $D_k^{\mathcal{BC}+}$.

Proof. By Definition 7, $D_k^{\mathcal{BC}+}$ is

$$0{:}Choice(\sigma_{\text{SF}}) \wedge \bigwedge_{(3.13) \in \mathcal{D}_{\mathcal{S}}} 0{:}G \rightarrow 0{:}F \wedge \bigwedge_{1 \leq i \leq k} \begin{array}{l} \bigwedge_{(3.13) \in \mathcal{D}_{\text{AD}}} (i-1){:}G \rightarrow (i-1){:}F \\ \wedge \bigwedge_{(3.13) \in \mathcal{D}_{\mathcal{S}}} i{:}G \rightarrow i{:}F \\ \wedge \bigwedge_{(3.14) \in \mathcal{D}_{\text{FD}}} (i-1){:}H \wedge i{:}G \rightarrow i{:}F \end{array}. \tag{B.1}$$

We show the proposition by induction on k as follows:
Base:

- $D_0^{\mathcal{BC}+}$ is (B.2) which is precisely (4.3).
- Therefore it follows trivially by definition that the answer sets $D_0^{\mathcal{BC}+}$ are exactly the histories of length 0 (i.e. states) of $\mathcal{T}(\mathcal{D}^{\mathcal{BC}+})$.

Inductive:

- Assume that $D_{k-1}^{\mathcal{BC}+}$ is a history of length $k-1$ of $\mathcal{T}(\mathcal{D}^{\mathcal{BC}+})$.
- The following can be easily observed:
  - it holds that

$$0{:}Choice(\sigma_{\mathrm{SF}}) \wedge \bigwedge_{(3.13)\in\mathcal{D}_{\mathcal{S}}} 0{:}G \rightarrow 0{:}F \tag{B.2}$$

    contains only atoms within $0{:}At(\sigma_{\mathrm{F}})$;
  - for each $1 \leq i \leq$ it holds that

$$\begin{array}{l} \bigwedge_{(3.13)\in\mathcal{D}_{\mathrm{AD}}} (i-1){:}G \rightarrow (i-1){:}F \\ \wedge \bigwedge_{(3.13)\in\mathcal{D}_{\mathcal{S}}} i{:}G \rightarrow i{:}F \\ \wedge \bigwedge_{(3.14)\in\mathcal{D}_{\mathrm{FD}}} (i-1){:}H \wedge i{:}G \rightarrow i{:}F \end{array} \tag{B.3}$$

    contains only atoms within $(i-1){:}At(\sigma_{\mathrm{F}}) \cup (i-1){:}At(\sigma_{\mathrm{A}}) \cup i{:}At(\sigma_{\mathrm{F}})$, and is negative on $(i-1){:}At(\sigma_{\mathrm{F}})$; and
  - finally, there are no rules in (B.1) such that for any $0 \leq i \leq j \leq k$
    - An action atom $a \in j{:}At(\sigma_{\mathrm{A}})$ occurs in the body, and a fluent atom $f \in i{:}At(\sigma_{\mathrm{F}})$ occurs in the head, and
    - A fluent atom $f \in j{:}At(\sigma_{\mathrm{F}})$ occurs in the body, and a fluent atom $f' \in i{:}At(\sigma_{\mathrm{F}})$ occurs in the head such that $i \neq j$.
- It follows by Module Theorem, that (B.1) is equivalent to

$$D_{k-1}^{\mathcal{BC}+} \sqcup \langle (\text{B.3}), (k-1){:}At(\sigma_{\mathrm{F}}), (k-1){:}At(\sigma_{\mathrm{A}}) \cup k{:}At(\sigma_{\mathrm{F}}) \rangle. \tag{B.4}$$

- Furthermore, by Module theorem, it holds that A is an answer set of (B.4) iff $A = A' \cup A_k$ such that

$$A' \subseteq 0{:}At(\sigma_{\mathrm{F}}) \cup \bigcup_{1\leq i<k} ((i-1){:}At(\sigma_{\mathrm{A}}) \cup i{:}At(\sigma_{\mathrm{F}})) \tag{B.5}$$

  is an answer set of $D_{k-1}^{\mathcal{BC}+}$ and

$$A \subseteq (k-1){:}At(\sigma_{\mathrm{A}}) \cup k{:}At(\sigma_{\mathrm{F}}) \tag{B.6}$$

  is an answer set of

$$\langle (\text{B.3}), (k-1){:}At(\sigma_{\mathrm{F}}), (k-1){:}At(\sigma_{\mathrm{A}}) \cup k{:}At(\sigma_{\mathrm{F}}) \rangle. \tag{B.7}$$

- It follows immediately from our induction hypothesis that A_{k-1} is a history of length $k-1$ of $\mathcal{T}(\mathcal{D}^{\mathcal{BC}+})$.

- Additionally, it can be observed that:
  - it holds that

$$\bigwedge_{(3.13)\in\mathcal{D}_{\mathrm{AD}}} (k-1){:}G \rightarrow (k-1){:}F \tag{B.8}$$

    contains only atoms within $(k-1){:}At(\sigma_{\mathrm{F}}) \cup (k-1){:}At(\sigma_{\mathrm{A}})$ and is negative on $(k-1){:}At(\sigma_{\mathrm{F}})$;
  - it holds that

$$\bigwedge_{(3.13)\in\mathcal{D}_{\mathcal{S}}} k{:}G \rightarrow k{:}F \tag{B.9}$$

    contains only atoms within $k{:}At(\sigma_{\mathrm{F}})$ and is negative on $k{:}At(\sigma_{\mathrm{SF}})$;
  - it holds that

$$\bigwedge_{(3.14)\in\mathcal{D}_{\mathrm{FD}}} (k-1){:}H \wedge k{:}G \rightarrow k{:}F \tag{B.10}$$

    contains only atoms within $(k-1){:}At(\sigma_{\mathrm{F}}) \cup (k-1){:}At(\sigma_{\mathrm{A}}) \cup k{:}At(\sigma_{\mathrm{F}})$ and is negative on $(k-1){:}At(\sigma_{\mathrm{F}}) \cup (k-1){:}At(\sigma_{\mathrm{A}}) \cup k{:}At(\sigma_{\mathrm{SD}})$.
  - finally, there are no rules in (B.3) such that a fluent atom $f \in k{:}At(\sigma_{\mathrm{F}})$ occurs in the body and action atom $a \in (k-1){:}At(\sigma_{\mathrm{A}})$ occurs in the head.
- It follows via Module Theorem that A_k is an answer set of (B.7) iff $A_k = A_{k-1,\mathrm{F}} \cup A_{k-1,\mathrm{A}} \cup A_{k,\mathrm{F}}$ such that $A_{k-1,\mathrm{A}} \subseteq (k-1){:}At(\sigma_{\mathrm{A}})$, $A_{k-1,\mathrm{F}} \subseteq (k-1){:}At(\sigma_{\mathrm{F}})$, and $A_{k,\mathrm{F}} \subseteq k{:}At(\sigma_{\mathrm{F}})$ where $A_{k-1,\mathrm{A}} \cup A_{k-1,\mathrm{F}}$ is an answer set of

$$\langle \bigwedge_{(3.13)\in\mathcal{D}_{\mathrm{AD}}} (k-1){:}G \rightarrow (k-1){:}F, (k-1){:}At(\sigma_{\mathrm{F}}), (k-1){:}At(\sigma_{\mathrm{A}})\rangle \tag{B.11}$$

  and A_k is an answer set of

$$\langle \bigwedge_{(3.14)\in\mathcal{D}_{\mathrm{FD}}} (k-1){:}H \wedge k{:}G \rightarrow k{:}F, (k-1){:}At(\sigma_{\mathrm{F}}) \cup (k-1){:}At(\sigma_{\mathrm{A}}), k{:}At(\sigma_{\mathrm{F}})\rangle. \tag{B.12}$$

- By definition, this is equivalent to $A_{k-1,\mathrm{A}} \cup A_{k-1,\mathrm{F}}$ being an answer set of (4.4) and A_k begin an answer set of (4.5).
- It only remains to observe that, by Lemma 2 it holds that A_k must also be an answer set of $k{:}(4.3) \wedge \mathit{Choice}((k-1){:}\sigma_{\mathrm{F}} \cup (k-1){:}\sigma_{\mathrm{A}})$, or, equivalently, $A_{k,\mathrm{F}}$ is an answer set of $k{:}(4.3)$.
- As A_{k-1} is a history of length $k-1$ of $\mathcal{T}(\mathcal{D}^{\mathcal{BC}+})$, it holds that $A_{k-1,\mathrm{F}}$ is a state of $\mathcal{T}(\mathcal{D}^{\mathcal{BC}+})$.
- It then follows that $A_{k-1,\mathrm{A}}$ is a candidate transition label leaving $A_{k-1,\mathrm{F}}$ and that A_k is a transition $\langle A_{k-1,\mathrm{F}}, A_{k-1,\mathrm{A}}, A_{k,\mathrm{F}}\rangle$ of $\mathcal{T}(\mathcal{D}^{\mathcal{BC}+})$.
- It follows immediately that $A = A_{k-1} \cup A_k$ is a history of length k of $\mathcal{T}(\mathcal{D}^{\mathcal{BC}+})$.

B.2 Proposition 3

Proposition 3
Given a definite $\mathcal{C}+$ action description $\mathcal{D}$, it holds that the transition system corresponding to $\mathcal{D}$ is exactly $\mathcal{T}(cp2bcp(\mathcal{D}))$.

Proof. Given a $\mathcal{C}+$ action description $\mathcal{D}$, by $\mathcal{D}_S$, $\mathcal{D}_{\mathrm{AD}}$ and $\mathcal{D}_{\mathrm{FD}}$ we denote the set of static, action dynamic, and fluent dynamic laws in $\mathcal{D}$.

Begin with the $\mathcal{C}+$ description $\mathcal{D}$. By definition, D_k is

$$0{:}Choice(\sigma_{\mathrm{SF}}) \wedge \bigwedge_{(3.9)\in\mathcal{D}_S} \neg\neg 0{:}G \to 0{:}F \wedge \bigwedge_{1\le i\le k} \begin{array}{l} \bigwedge_{(3.9)\in\mathcal{D}_{\mathrm{AD}}} \neg\neg(i-1){:}G \to (i-1){:}F \\ \wedge \bigwedge_{(3.9)\in\mathcal{D}_{\mathcal{S}}} \neg\neg i{:}G \to i{:}F \\ \wedge \bigwedge_{(3.10)\in\mathcal{D}_{\mathrm{FD}}} (i-1){:}H \wedge \neg\neg i{:}G \to i{:}F \end{array} . \tag{B.13}$$

This is exactly the propositional formula generated by $cp2bcp(\mathcal{D})$ under the $\mathcal{BC}+$.

B.3 Proposition 4

Proposition 4
Given a $\mathcal{BC}$ action description $\mathcal{D}$, it holds that the transition system corresponding to $\mathcal{D}$ is exactly $\mathcal{T}(bc2bcp(\mathcal{D}))$.

Proof. Given a $\mathcal{BC}$ action description $\mathcal{D}$, by $\mathcal{D}_S$ and $\mathcal{D}_{\mathrm{FD}}$ we denote the set of static and fluent dynamic laws in $\mathcal{D}$.

Begin with the $\mathcal{BC}$ description $\mathcal{D}$. By definition, D_k is

$$\begin{array}{l} 0{:}Choice(\sigma_{\mathrm{SF}}) \wedge \bigwedge_{(3.13)\in\mathcal{D}_S} 0{:}G_1 \wedge \neg\neg 0{:}G_2 \to 0{:}F \\ \qquad \wedge \bigwedge_{1\le i\le k} \begin{array}{l} \bigwedge_{(3.13)\in\mathcal{D}_S} i{:}G_1 \wedge \neg\neg i{:}G_2 \to i{:}F \\ \wedge (i-1){:}Choice(\sigma_{\mathrm{A}}) \\ \wedge \bigwedge_{(3.14)\in\mathcal{D}_{\mathrm{FD}}} (i-1){:}H \wedge i{:}G_1 \wedge \neg\neg i{:}G_2 \to i{:}F \end{array} . \end{array} \tag{B.14}$$

This is exactly the propositional formula generated by $bc2bcp(\mathcal{D})$ under the $\mathcal{BC}+$.

B.4 Theorem 3

Theorem 4 *Let F, G, and H be first-order sentences, and let $\mathbf{p}$, $\mathbf{q}$ be finite lists of distinct predicate constants. If*

(a) each strongly connected component of the predicate dependency graph of $F \wedge G \wedge H$ relative to $\mathbf{p}$, $\mathbf{q}$ is a subset of $\mathbf{p}$ or a subset of $\mathbf{q}$,
(b) F is negative on $\mathbf{q}$, and
(c) G is negative on $\mathbf{p}$

then

$$\mathrm{SM}[F \wedge G \wedge H;\ \mathbf{pq}] \leftrightarrow \mathrm{SM}[F \wedge H;\ \mathbf{p}] \wedge \mathrm{SM}[G \wedge H;\ \mathbf{q}]$$

is logically valid.

Proof. Let $\mathbf{p}'$ ($\mathbf{q}'$) be the set of intensional predicates which are exclusive to $\mathbf{p}$ ($\mathbf{q}$), and $\mathbf{r}$ be be the intersection of $\mathbf{p}, \mathbf{q}$. We have the following chain of equivalences via Ferraris *et al.* (2009b): Splitting Lemma Version 3

$$\begin{aligned}
&\mathrm{SM}[F \wedge G \wedge H; \mathbf{pq}] && \leftrightarrow \\
&\mathrm{SM}[F \wedge G \wedge H; \mathbf{p}'\mathbf{q}'\mathbf{r}] && \leftrightarrow \\
&\mathrm{SM}[F \wedge G \wedge H; \mathbf{p}^{1'}] \wedge \cdots \wedge \mathrm{SM}[F \wedge G \wedge H; \mathbf{p}^{n'}] \\
&\wedge \mathrm{SM}[F \wedge G \wedge H; \mathbf{q}^{1'}] \wedge \cdots \wedge \mathrm{SM}[F \wedge G \wedge H; \mathbf{q}^{n'}] \\
&\wedge \mathrm{SM}[F \wedge G \wedge H; \mathbf{r}^{1}] \wedge \cdots \wedge \mathrm{SM}[F \wedge G \wedge H; \mathbf{r}^{n}] && \leftrightarrow \\
&\mathrm{SM}[F \wedge G \wedge H; \mathbf{p}'] \wedge \mathrm{SM}[F \wedge G \wedge H; \mathbf{q}'] \wedge \mathrm{SM}[F \wedge G \wedge H; \mathbf{r}] && \leftrightarrow \\
&\mathrm{SM}[F \wedge G \wedge H; \mathbf{p}'\mathbf{r}] \wedge \mathrm{SM}[F \wedge G \wedge H; \mathbf{q}'\mathbf{r}] && \leftrightarrow \\
&\mathrm{SM}[F \wedge G \wedge H; \mathbf{p}] \wedge \mathrm{SM}[F \wedge G \wedge H; \mathbf{q}] && \leftrightarrow \\
&\mathrm{SM}[F \wedge H; \mathbf{p}] \wedge G \wedge \mathrm{SM}[G \wedge H; \mathbf{q}] \wedge F.
\end{aligned}$$

This is then equivalent to

$$\mathrm{SM}[F \wedge H; \mathbf{p}] \wedge \mathrm{SM}[G \wedge H; \mathbf{q}].$$

as $\mathrm{SM}[F \wedge H; \mathbf{p}]$ ($\mathrm{SM}[G \wedge H; \mathbf{q}]$) entails F (G) by the definition of the SM operator.

Lemma 3 *Given a second-order sentence F and compatible partial interpretations A_1, A_2 such that all constants occurring in F are in $\mathbf{c}_1$ and $\mathbf{c}_2$. It holds that $A_1 \models F$ iff $A_2 \models F$.*

Proof. We simply must show that $F^{A_1} = F^{A_2}$. By Induction.

Base: F is an atomic formula $p(t_1, ..., t_n)$ where $t_1, ..., t_n$ are variable-free terms, equality $t_1 = t_2$ where t_1 and t_2 are variable free terms, or $\bot$.

If F is an atomic formula, we have that $F^{A_1} = p^{A_1}(t_1^{A_1}, ..., t_n^{A_1}) = p^{A_1}(t_1^{A_2}, ..., t_n^{A_2})$ from 4, it then follows directly that this is $p^{A_2}(t_1^{A_2}, ..., t_n^{A_2}) = F^{A_2}$ as A_1 and A_2 are compatible and $p \in \mathbf{c}_1 \cap \mathbf{c}_2$.

Similarly, if F is an equality, then F^{A_1} is $t_1^{A_1} = t_2{}_1^{A}$ which is then $t_1^{A_2} = t_2^{A_2}$ or equivalently F^{A_2}, by Lemma 4.

If F is $\bot$, then $F^{A_1} = F^{A_2} = \mathtt{f}$ trivially.

Inductive: Let Lemma 3 hold for any second-order sentences which are shorter than F. We show that the Lemma holds for the case where F is $G \odot H$ ($\odot \in \{\wedge, \vee, \rightarrow\}$), F is $QxG(x)$ where $Q \in \{\forall, \exists\}$ and x is a predicate-variable, and F is $QxG(x)$ where $Q \in \{\forall, \exists\}$ and x is a object-variable.

If F is $G \odot H$, then

$$\begin{aligned}
F^{A_1} &= \odot(G^{A_1}, H^{A_1}) \\
&= \odot(G^{A_2}, H^{A_2}) = F^{A_2}. && \text{by the I.H.}
\end{aligned}$$

If F is $QxG(x)$ such that x is a predicate-variable, it is sufficient to show that each resulting substitution $G(\rho)$, where ρ is an arbitrary predicate name from the extended

signature of the same arity as x, is evaluated similarly in each interpretation. As $G(\rho)$ is shorter than F, this holds by the induction hypthesis.

Finally, If F is $QxG(x)$ such that x is an object-variable, we have that $|A_1| = |A_2|$ so the set of possible object constants of the extended signature mapping to the universe is identical for each interpretation. It also follows from the I.H. that $G(\xi^*)^{A_1} = G(\xi^*)^{A_2}$ for each such object constant ξ^*. It then holds that $(QxG(x))^{A_1} = (QxG(x))^{A_2}$.

Therefore, it holds that $A_1 \models F$ iff $A_2 \models F$.

Lemma 4 *Given a variable-free term t and compatible partial interpretations A_1, A_2, such that all constants occurring within t are within $\mathbf{c}_1 \cap \mathbf{c}_2$. It holds that $t^{A_1} = t^{A_2}$.*

Proof. By Induction.

Base: t is an object constant: As A_1 and A_2 are compatible and $t \in \mathbf{c}_1 \cap \mathbf{c}_2$, it holds that $t^{A_1} = t^{A_2}$.

Inductive: Let $t_1, ..., t_n$ be variable-free terms such that $t_i^{A_1} = t_i^{A_2}$ for each $1 \leq i \leq n$. If $t = t_0(t_1, ..., t_n)$, then

$$\begin{aligned} t^{A_1} &= t_0^{A_1}(t_1^{A_1}, ..., t_n^{A_1}) && \text{Definition} \\ &= t_0^{A_1}(t_1^{A_2}, ..., t_n^{A_2}) && \text{I.H.} \\ &= t_0^{A_2}(t_1^{A_2}, ..., t_n^{A_2}) && \text{Compatible Interpretations} \\ &= t_0^{A_2}. && \text{Definition} \end{aligned}$$

Theorem 3
Let $\mathbb{F}_1 = \langle F_1, I_1, O_1 \rangle$ and $\mathbb{F}_2 = \langle F_2, I_2, O_2 \rangle$ be first-order modules that are joinable with interpretations A_1 and A_2 of $\mathbf{c}_1 \supseteq \mathbf{c}(F_1) \cup O_1$ and $\mathbf{c}_2 \supseteq \mathbf{c}(F_2) \cup O_2$, respectively. If I_1 A and A_2 are compatible with each other,

$$A_1 \cup A_2 \models \mathrm{SM}[\mathbb{F}_1 \sqcup \mathbb{F}_2] \quad \textit{iff} \quad A_1 \models \mathrm{SM}[\mathbb{F}_1] \ \textit{ and } \ A_2 \models \mathrm{SM}[\mathbb{F}_2] \ .$$

Proof. Let $\mathbb{F}_1 = \langle F_1' \wedge H, I_1, O_1 \rangle$ and $\mathbb{F}_2 = \langle F_2' \wedge H, I_2, O_2 \rangle$.

By definition $\mathrm{SM}[\mathbb{F}_1 \sqcup \mathbb{F}_2]$ is $\mathrm{SM}[F_1'' \wedge F_2'' \wedge H; O_1 \cup O_2]$. By Theorem 4, it holds that

$$\begin{aligned} &A_1 \cup A_2 \models \mathrm{SM}[F_1' \wedge F_2' \wedge H; O_1 \cup O_2] \text{ iff} \\ &\qquad A_1 \cup A_2 \models \mathrm{SM}[F_1' \wedge H; O_1] \text{ and } A_1 \cup A_2 \models \mathrm{SM}[F_2' \wedge H; O_2] \end{aligned}$$

It can trivially be shown that $A_1 \cup A_2$ is compatible with A_1, it follows from our assumptions and by Lemma 3 that $A_1 \cup A_2 \models \mathrm{SM}[F_1' \wedge H; O_1]$ if, and only if, $A_1 \models \mathrm{SM}[F_1' \wedge H; O_1]$. Similarly, $A_1 \cup A_2 \models \mathrm{SM}[F_1' \wedge H; O_1]$ if, and only if, $A_2 \models \mathrm{SM}[F_2' \wedge H; O_1]$. It follows that

$$\begin{aligned} &A_1 \cup A_2 \models \mathrm{SM}[\mathbb{F}_1 \sqcup \mathbb{F}_2] \text{ iff} \\ &\qquad A_1 \models \mathrm{SM}[\mathbb{F}_1] \text{ and } A_2 \models \mathrm{SM}[\mathbb{F}_2] \end{aligned}$$

B.5 Proposition 7

Proposition 7
Given a modular incremental theory $\langle B, P[t], Q[t]\rangle$ *and online progression* $\langle E, F\rangle_{\geq 1}$ *and any* $j, k \geq 0$ *such that* $e_1, \ldots, e_j, f_j \leq k$*, an interpretation* I *is an answer set of* $\mathbb{R}_{j,k}$ *iff there are compatible interpretations*

$$I_B, I_{P[t/1]}, \ldots, I_{P[t/k]}, I_{E_1[e_1]}, \ldots, I_{E_j[e_j]}, I_{Q[t/k]}, I_{F_j[f_j]} \tag{B.15}$$

such that $I = \bigcup_{X \in (\text{B.15})} X$ *where*

- I_B *is an answer set of* $PM(B, I(B))$*,*
- *each* $I_{P[t/i]}$ *is an answer set of* $PM(P[t/i], O(\mathbb{P}_{i-1}) \cup I(P[t/i]))$*,*
- *each* $I_{E_i[e_i]}$ *is an answer set of* $PM(E_i[e_i], O(\mathbb{P}_{e_i}) \cup O(\mathbb{E}_{i-1}) \cup I(E_i[e_i]))$*,*
- $I_{Q[t/k]}$ *is an answer set of* $PM(Q[k/t], O(\mathbb{P}_k) \cup I(Q[t/k]))$*, and*
- $I_{F_j[f_j]}$ *is an answer set of* $PM(F_j[f_j], O(\mathbb{P}_{f_j}) \cup O(\mathbb{E}_j) \cup I(F_j[f_j]))$*.*

Proof. As the incremental theory and online progression are modular, each of the modules provided in Definition 16 are defined and joinable. All that is required is to inductively apply the module theorem on the structure provided therein.

B.6 Proposition 8

Given a formula F, by $Head(F)$ we denote the set of predicates which have strictly-positive occurrences in F.

Lemma 5 *Given propositional formulas* F*,* G *and set of atoms* I *such that* G *contains no strictly-positive occurrences of* $Pred(F) \setminus I$ *it holds that the stable models of* $F \wedge G$ *and* $Simple(F, I) \wedge G$ *coincide.*

Proof. By induction on the number of iterations of $Simple(F, I)$.

base

Iteration 0: Trivial as $F \wedge G$ clearly has the same stable models as itself.

inductive

- Let F_{i-1} be the formula obtained by performing $i - 1$ iterations of the *Simple* operation and assume that $F_{i-1} \wedge G$ and $F \wedge G$ have the same stable models.
- It is clear that $Pred(F_{i-1}) \subseteq Pred(F)$.
- As G contains no strictly positive occurrences of atoms in $Pred(F) \setminus I$, it follows that the same holds for $Pred(F_{i-1}) \setminus I$.
- Therefore $F_{i-1} \wedge G$ contains no strictly positive occurrences of atoms within $X = Pred(F_{i-1}) \setminus (Head(F_{i-1}) \cup I)$.
- Let F'_{i-1} be the propositional formula obtained from F_{i-1} by replacing occurrences of all atoms $a \in X$ with $\bot$.

- By Ferraris *et al.* (2011) Theorem 4, it then holds that $F_{i-1} \wedge G$ and $F'_{i-1} \wedge G$ have the same stable models.
- Finally, it is only necessary to observe that the transformations prescribed during projection are strong equivalencies.
- This establishes that the stable models of $F'_{i-1} \wedge G$ and $F_i \wedge G$ coincide.
- By the I.H. it then holds that the stable models of $F \wedge G$ and $F_i \wedge G$ coincide.

Proposition 8
Given an incremental theory $\langle B, P[t], Q[t] \rangle$ and online progression $\langle E, F \rangle_{\geq 1}$ which are modular and mutually revisable and some $j, k \geq 0$ such that $e_1, \ldots, e_j, f_j \leq k$ and let $R_{j,k}$ and $\mathbb{R}_{j,k} = \langle R, I, O \rangle$ be the k-expansion and incremental composition, respectively. It holds that the stable models of $R_{j,k}$ and R coincide.

Proof.
For each $G \in (5.6)$, we define I_G such that

$$\begin{aligned} I_B &= I(B), \\ I_{P[t/i]} &= O(\mathbb{P}_{i-1}) \cup I(P[t/i]), \\ I_{E_i[e_i]} &= O(\mathbb{P}_{e_i}) \cup O(\mathbb{E}_{i-1}) \cup I(E_i[e_i]), \\ I_{Q[t/i]} &= O(\mathbb{P}_i) \cup I(Q[t/i]), \text{ and} \\ I_{F_i[f_i]} &= O(\mathbb{P}_{f_i}) \cup O(\mathbb{E}_i) \cup I(F_j[f_j]). \end{aligned}$$

Equivalently, this can be represented as

$$I_G = I(G) \cup \bigcup_{G' \prec G} \mathit{Simple}(G, I_{G'}) \setminus I(G').$$

By definition, $R_{j,k}$ is

$$B \wedge P[t/1] \wedge \cdots \wedge P[t/k] \wedge E_1[e_1] \wedge \cdots \wedge E_j[e_j] \wedge Q[t/k] \wedge F_j[f_j].$$

For uniformity, we refer to each formula $G \in (5.6)$ by its index in the conjunction, making $R_{j,k}$

$$G_1 \wedge G_2 \wedge \cdots \wedge G_{k+1} \wedge G_{k+2} \wedge \cdots \wedge G_{j+k+1} \wedge G_{j+k+2} \wedge G_{j+k+3} \tag{B.16}$$

Observe that for each pair of formulas G_{i_1}, and G_{i_2} such that $i_2 > i_1$ it holds that $G_{i_2} \not\prec G_{i_1}$.

As the theory is mutually revisable, it then holds by definition that for any $i_2 > i_1$, G_{i_2} is negative on $\mathit{Pred}(G_{i_1}) \setminus I(G_{i_1})$. Furthermore, it can be observed that $\mathit{Simple}(G_{i_1}, I_{G_{i_1}})$ is negative on $\mathit{Pred}(G_{i_2}) \setminus I_{G_{i_2}}$:

- If $G_{i_1} \prec G_{i_2}$ it can be observed that

  $$\mathit{Pred}(\mathit{Simple}(G_{i_1}, I_{G_{i_1}})) \setminus I_(G_{i_1}) \subseteq I_{G_{i_2}}.$$

  As G_{i_1} is negative on $I(G_{i_1})$ it then follows that each strictly positively occurring atom in G_{i_1} is within $I_{G_{i_2}}$. Therefore, $\mathit{Simple}(G_{i_1}, I_{G_{i_1}})$ is negative on $\mathit{Pred}(G_{i_2}) \setminus I_{G_{i_2}}$.

- Otherwise $G_{i_1} \not\prec G_{i_2}$. As the theory is mutually revisable, it holds by definition that G_{i_1} (and therefore $\mathit{Simple}(G_{i_1}, I_{G_{i_1}})$) is negative on $\mathit{Pred}(G_{i_e}) \setminus I(G_{i_2})$.

It follows via repeated applications of Lemma 5 that (B.16) has the same stable models as

$$\mathit{Simple}(G_1, I_{G_1}) \wedge \mathit{Simple}(G_2, I_{G_2}) \wedge \cdots \wedge \mathit{Simple}(G_{j+k+3}, I_{G_{j+k+3}}).$$

By taking F to be each G_i and G to be

$$\mathit{Simple}(G_1, I_{G_1}) \wedge \cdots \wedge \mathit{Simple}(G_{i-1}, I_{G_{i-1}}) \wedge G_{i+1} \wedge \cdots \wedge G_{j+k+3}.$$

on each iteration.

This is exactly R.

B.7 Lemma 1

Lemma 6 *Given propositional formulas F, G and disjoint sets of atoms A_1, A_2 such that each atom occurring outside the scope of negation within F or G is contained in A_1 or A_2, respectively. It holds that* $\mathrm{DG}[F \wedge G; A_1 \cup A_2]$ *contains no cycles spanning A_1 and A_2.*

Proof.

- Assume the presence of such a cycle.
- It follows that there is then some strictly positive implication of the form $H_1 \rightarrow H_2$ such that some $p \in A_1$ occurs positively and outside the scope of negation in H_1 and some $q \in A_2$ occurs strictly positively in H_2.
- Note that $H_1 \rightarrow H_2$ contains occurrences of atoms within both A_1 and A_2 outside the scope of negation.
- $H_1 \rightarrow H_2$ cannot occur in F as each atom occurring outside negation in F is in A_1, which is disjoint from A_2.
- Similarly, $H_1 \rightarrow H_2$ cannot occur in G as G cannot have a non-negated occurrence of an atom within A_1.
- Thus a contradiction is drawn.

Lemma 7 *Given an acyclic incremental theory $\langle B, P[t], Q[t]\rangle$ and online progression $\langle E, F\rangle_{\geq 1}$, it holds that, for any $i \geq 0$, $\mathbb{P}_i$ is defined such that*

- *the set of atoms occurring within $F(\mathbb{P}_i)$ outside negation are contained within $O(\mathbb{P}_i)$, and*
- *$O(\mathbb{P}_i) \subseteq O_{P_i}$ where O_{P_i} is*

$$\mathit{Head}(B) \cup \mathit{Head}(P[t/1]) \cup \cdots \cup \mathit{Head}(P[t/i]).$$

Proof. By induction on i:

Base ($i = 0$)**:**

- Trivially

$$\mathbb{P}_0 = \langle \mathit{Simple}(B, I(B)), I(B), \mathit{Pred}(\mathit{Simple}(B, I(B))) \setminus I(B) \rangle$$

  is defined.
- As the theory is acyclic, B contains no occurrences of atoms within $I(B)$ outside the scope of negation.
- It follows that the set of atoms occurring within $F(\mathbb{P}_0)$ outside negation are contained within $O(\mathbb{P}_0)$
- Additionally, by definition, $\mathit{Pred}(\mathit{Simple}(B, I(B))) \subseteq \mathit{Head}(B) \cup I(B)$.
- It follows trivially that $O(\mathbb{P}_0) \subseteq \mathit{Head}(B)$.

Inductive ($i > 0$)**:** Assume the lemma holds for $i - 1$.

- Let $I = I(P[t/i]) \cup O(\mathbb{P}_{i-1})$.
- Trivially

$$\mathbb{P}[t/i] = \langle \mathit{Simple}(P[t/i], I), I, \mathit{Pred}(\mathit{Simple}(P[t/i], I)) \setminus I \rangle$$

  is defined.
- By definition, $O(\mathbb{P}[t/i])$ and $O(\mathbb{P}_{i-1})$ are disjoint.
- Additionally, it holds that there all atoms occurring within $F(\mathbb{P}[t/i])$ outside the scope of negation are within $O(\mathbb{P}[t/i])$:
  - As the incremental theory is acyclic, $P[t/i] \not\prec B$, and $P[t/i] \not\prec P[t/j]$ ($j < i$), it holds that $P[t/i]$ contains no occurrences of any atoms within

$$\begin{aligned}(\mathit{Pred}(B) \setminus I(B)) \cup (\mathit{Pred}(P[t/1]) \setminus I(P[t/1])) \cup \ldots \\ \cup (\mathit{Pred}(P[t/i-1]) \setminus I(P[t/i-1]))\end{aligned} \tag{B.17}$$

    outside the scope of negation.
  - By the induction hypothesis, (B.17) is a super set of $O(\mathbb{P}_{i-1})$.
  - Additionally, as the incremental theory is acyclic, $P[t/i]$ contains only negated occurrences of any atoms within $I(P[t/i])$.
  - Since $I = I(P[t/i]) \cup O(\mathbb{P}_{i-1})$, it then follows that $F(\mathbb{P}[t/i])$ contains no non-negated occurrences of any atoms within I.
  - Therefore non-negated atoms within $F(\mathbb{P}[t/i])$ must be in $O(\mathbb{P}[t/i])$.
- It then follows by the induction hypothesis and Lemma 6 that

$$\mathrm{DG}[F(\mathbb{P}_{i-1}) \wedge F(\mathbb{P}[t/i]); O(\mathbb{P}_{i-1}) \cup O(\mathbb{P}[t/i])]$$

  has no cycles spanning $O(\mathbb{P}_{i-1})$ and $O(\mathbb{P}[t/i])$.

- It follows immediately that

$$\mathbb{P}_i = \mathbb{P}_{i-1} \sqcup \mathbb{P}[t/i]$$

  is defined.
- Furthermore, from the definition of *Simple*, it can trivially be seen that

$$O(\mathbb{P}[t/i]) \subseteq \mathit{Head}(P[t/i]).$$

- It then holds by the induction hypothesis that

$$O(\mathbb{P}_i) = O(\mathbb{P}_{i-1}) \cup O(\mathbb{P}[t/i]) \subseteq O_{P_{i-1}} \cup \mathit{Head}(P[t/i]) = O_{P_i}.$$

- Finally, it is only necessary to observe that as each atom occurring within $F(\mathbb{P}_{i-1})$ and $F(\mathbb{P}[t/i])$ outside the scope of negation is in $O(\mathbb{P}_{i-1})$ and $O(\mathbb{P}[t/i])$, respectively, it follows that each atom occurring within $F(\mathbb{P}_i)$ is within $O(\mathbb{P}_i)$.

Lemma 8 *Given an acyclic incremental theory $\langle B, P[t], Q[t]\rangle$ and online progression $\langle E, F\rangle_{\geq 1}$, it holds that, for any $i \geq 0$, $\mathbb{E}_i$ is defined such that*

- *the set of atoms occurring within $F(\mathbb{E}_i)$ outside negation are contained within $O(\mathbb{E}_i)$, and*
- *$O(\mathbb{E}_i) \subseteq O_{E_i}$ where O_{E_i} is*

$$\mathit{Head}(E_1[e_1]) \cup \cdots \cup \mathit{Head}(E_i[e_i]).$$

Proof. By induction on i:

Base ($i = 0$)**:**

- Trivial.

Inductive ($i > 0$)**:** Assume the lemma holds for $i - 1$.

- By Lemma 7, $\mathbb{P}_{e_i}$ is defined.
- Let $I = I(E_i[e_i]) \cup O(\mathbb{E}_{i-1}) \cup O(\mathbb{P}_{e_i})$.
- Trivially

$$\mathbb{E}_i[e_i] = \langle \mathit{Simple}(E_i[e_i], I), I, \mathit{Pred}(\mathit{Simple}(E_i[e_i], I)) \setminus I\rangle$$

  is defined.
- By definition, $O(\mathbb{E}_i[e_i])$ and $O(\mathbb{E}_{i-1})$ are disjoint.
- Additionally, it holds that there all atoms occurring within $F(\mathbb{E}_i[e_i])$ outside the scope of negation are within $O(\mathbb{E}[e_i])$:

– As the incremental theory is acyclic, $E_i[e_i] \not\prec B$, $E_i[e_i] \not\prec P[t/j]$ $(j \leq e_j)$, and $E_i[e_i] \not\prec E_j[e_j]$ $(j < i)$, it holds that $E_i[e_i]$ contains no non-negated occurrences of any atoms within

$$(\mathit{Pred}(B) \setminus I(B)) \cup (\mathit{Pred}(P[t/1]) \setminus I(P[t/1])) \cup \cdots \cup (\mathit{Pred}(P[t/e_i]) \setminus I(P[t/e_i])) \tag{B.18}$$

$$\cup (\mathit{Pred}(E_1[e_1]) \setminus I(E_1[e_1])) \cup \cdots \cup (\mathit{Pred}(E_i[e_i]) \setminus I(E_i[e_i])) \tag{B.19}$$

– By the induction hypothesis, (B.19) is a super set of $O(\mathbb{E}_{i-1}) \cup O(\mathbb{P}_{e_i})$.
– Additionally, as the incremental theory is acyclic, $E_i[e_i]$ contains no non-negated occurrences of any atoms within $I(E_i[e_i])$.
– It then follows that $P[t/i]$ contains only negated occurrences of any atoms within I.
– Therefore non-negated atoms within $E_i[e_i]$ must be in $O(\mathbb{E}_i[e_i])$.

- It then follows by the induction hypothesis and Lemma 6 that

$$\mathrm{DG}[F(\mathbb{E}_{i-1}) \wedge F(\mathbb{E}_i[e_i]); O(\mathbb{E}_{i-1}) \cup O(\mathbb{E}_i[e_i])]$$

  has no cycles spanning $O(\mathbb{E}_{i-1})$ and $O(\mathbb{E}_i[e_i])$.

- It follows immediately that

$$\mathbb{E}_i = \mathbb{E}_{i-1} \sqcup \mathbb{E}_i[e_i]$$

  is defined.

- Furthermore, from the definition of *Simple*, it can trivially be seen that

$$O(\mathbb{E}_i[e_i]) \subseteq \mathit{Head}(E_i[e_i]).$$

- It then holds by the induction hypothesis that

$$O(\mathbb{E}_i) \subseteq O_{E_{i-1}} \cup \mathit{Head}(E_i[e_i]) = O_{E_i}.$$

- Finally, it is only necessary to observe that as each atom occurring within $F(\mathbb{E}_{i-1})$ and $F(\mathbb{E}_i[e_i])$ outside the scope of negation is in $O(\mathbb{E}_{i-1})$ and $O(\mathbb{E}_i[e_i])$, respectively, it follows that each atom occurring within $F(\mathbb{E}_i)$ is within $O(\mathbb{E}_i)$.

Lemma 9 *Given an acyclic incremental theory $\langle B, P[t], Q[t]\rangle$ and online progression $\langle E, F\rangle_{\geq 1}$, it holds that they are modular.*

Proof.
We show that for any $j, k \geq 0$ such that $e_1, \ldots, e_j, f_j \leq k$, $\mathbb{R}_{j,k}$ is defined:

- By Lemmas 7 and 8, $\mathbb{P}_k$ and $\mathbb{E}_j$ are defined.
- Furthermore, as the theory is acyclic, it holds that O_{P_k} and O_{E_j} are disjoint:

  - Assume that there is some $a \in O_{P_k} \cap O_{E_j}$.
  - By definition, it then holds that there is some $F \in \{B, P[t/1], \ldots, P[t/k]\}$ and some $G \in \{E_1[e_1], \ldots, E_j[e_j]\}$ such that $a \in \mathit{Head}(F) \cap \mathit{Head}(G)$.
  - Therefore, it holds that F and G both contain an occurrence of a outside the scope of negation.
  - Observe that for any choices of F and G, $G \not\prec F$.
  - Therefore, as the theory is acyclic, G cannot contain a non-negated occurrence of a as, clearly, $a \in \mathit{Pred}(F) \setminus I(F)$.
  - This contradicts the fact that $a \in \mathit{Head}(G)$.
- Therefore, it holds from Lemmas 7 and 8 that $O(\mathbb{P}_k)$ and $O(\mathbb{E}_j)$ are disjoint.
- It then follows from Lemma 6 that there are no loops in

$$\mathrm{DG}[F(\mathbb{P}_k) \wedge F(\mathbb{E}_j); O(\mathbb{P}_k) \cup O(\mathbb{E}_j)].$$

- It follows immediately that

$$\mathbb{M}_{j,k} = \mathbb{P}_k \sqcup \mathbb{E}_j$$

  is defined.
- As $O(\mathbb{P}_k) \subseteq O_{P_k}$ and $O(\mathbb{E}_j) \subseteq O_{E_j}$ it holds that $O(\mathbb{M}_{j,k}) \subseteq O_{P_k} \cup O_{E_j}$.
- Additionally, as the set of atoms occurring within $F(\mathbb{P}_k)$ outside negation are exactly the atoms in $O(\mathbb{P}_k)$ (and similarly for $F(\mathbb{E}_j)$ and $O(\mathbb{E}_j)$), it can be trivially observed that the set of atoms occurring within $F(\mathbb{M}_{j,k})$ outside negation are exactly the atoms in $O(\mathbb{M}_{j,k})$.

- Next, it can be trivially seen that

$$\mathbb{Q}[t/k] = \langle \mathit{Simple}(Q[t/k], I_Q), I_Q, \mathit{Pred}(\mathit{Simple}(Q[t/k]) \setminus I_Q \rangle$$

  is defined where $I_Q = I(Q[t/k]) \cup O(\mathbb{P}_k)$.
- From the definition of *Simple* we have that $O(\mathbb{Q}[t/k]) \subseteq \mathit{Head}(Q[t/k])$.
- As the theory is acyclic, it can be seen quite easily from the definition of $\prec$ that only atoms in $O(\mathbb{Q}[t/k])$ have non-negated occurrences within $F(\mathbb{Q}[t/k])$.
- Furthermore, it can be observed that $O(\mathbb{Q}[t/k])$ is disjoint from $O(\mathbb{M}_{j,k})$:
  - Assume there is some $a \in O(\mathbb{Q}[t/k]) \cap O(\mathbb{M}_{j,k})$.
  - It's clear from the definition of $O(\mathbb{Q}[t/k])$ that $a \notin O(\mathbb{P}_k)$, therefore it holds that $a \in O(\mathbb{E}_j)$.
  - By definition, there is then some $F \in \{E_1[e_1], \ldots, E_j[e_j]\}$ such that $a \in \mathit{Head}(F) \cap \mathit{Head}(Q[t/k])$.
  - Similar to the case for O_{P_k} and O_{E_j}, this then contradicts the acyclic assumption as, clearly, $F \not\prec Q[t/k]$.
- It then follows from Lemma 6 that there are no loops in the joint dependency graph spanning both outputs.

- It follows immediately that

$$\mathbb{N}_{j,k} = \mathbb{M}_{j,k} \sqcup \mathbb{Q}[t/k]$$

  is defined.
- Similar to $\mathbb{M}_{j,k}$, it promptly follows that

$$O(\mathbb{N}_{j,k}) \subseteq \mathit{Head}(Q[t/k]) \cup O_{P_k} \cup O_{E_j}$$

  and that the set of atoms occurring within $F(\mathbb{N}_{j,k})$ outside negation are exactly the atoms in $O(\mathbb{N}_{j,k})$.

- Finally, it can be trivially seen that

$$\mathbb{F}_j[f_j] = \langle \mathit{Simple}(F_j[f_j], I_F), I_F, \mathit{Pred}(\mathit{Simple}(F_j[f_j]) \setminus I_F \rangle$$

  is defined where $I_F = I(F_j[f_j]) \cup O(\mathbb{E}_j) \cup O(\mathbb{P}_{f_j})$.
- From the definition of *Simple* we have that $O(\mathbb{F}_j[f_j]) \subseteq \mathit{Head}(F_j[f_j])$.
- As the theory is acyclic, it can bee seen quite easily from the definition of $\prec$ that only atoms in $O(\mathbb{F}_j[f_j])$ have non-negated occurrences within $F(\mathbb{F}_j[f_j])$.
- Furthermore, it can be observed that $O(\mathbb{F}_j[f_j])$ is disjoint from $O(\mathbb{N}_{j,k})$:
  - Assume there is some $a \in O(\mathbb{Q}[t/k]) \cap O(\mathbb{M}_{j,k})$.
  - By definition, there is then some

$$F \in \{B, P[t/1], \ldots, P[t/k], E_1[e_1], \ldots, E_j[e_j], Q[t/k]\}$$

    such that $a \in \mathit{Head}(F) \cap \mathit{Head}(F_j[f_j])$.
  - Similar to the previously, this then contradicts the acyclic assumption as, clearly, $F_j[f_j] \not\prec F$.
- It then follows from Lemma 6 that there are no loops in the joint dependency graph spanning both outputs.
- It follows immediately that

$$\mathbb{R}_{j,k} = \mathbb{N}_{j,k} \sqcup \mathbb{F}_j[f_j]$$

  is defined.

Lemma 10 *Given an acyclic incremental theory $\langle B, P[t], Q[t] \rangle$ and online progression $\langle E, F \rangle_{\geq 1}$, it holds that they are mutually revisable.*

Proof.

- Let F, G be distinct coexisting incremental components of the incremental theory and online progression such that $F \not\prec G$.
- As the theory is acyclic, it holds that each atom $a \in \mathit{Pred}(G) \setminus I(G)$ only occurs within F in the scope of negation.
- It follows immediately that $\mathit{Head}(F) \cap \mathit{Pred}(G) \setminus I(G) = \emptyset$.
- Therefore, the theory is mutually revisable.

Lemma 1
Given an acyclic incremental theory $\langle B, P[t], Q[t] \rangle$ and online progression $\langle E, F \rangle_{\geq 1}$, it holds that they are modular and mutually revisable.

Proof. Immediate from Lemmas 9 and 10.

B.8 Proposition 9

Lemma 11 *Given any propositional formula F, if an atom occurs outside the scope of negation in prop2dlf(F), then there is a corresponding occurrence outside the scope of negation in F.*

Proof. By induction on the number n of transformations performed on F.
Base $(n = 0)$**:** Trivial.
Inductive $(n \geq 1)$**:** Assume that the lemma holds for $n - 1$ transformations. We show that for each transformation if a (non-trivial) subformula occurs within the scope of negation prior to the transformation, then it occurs within the scope of negation afterward.

Among the possible transformations, only (6.2)-(6.7), (6.14), and (6.20) have subformulas which occur within the scope of negation. Among these, (6.2) and (6.3) are trivial. We consider the remainder:

(6.4)**:** G occurs within 3 negations, afterward it occurs within 1.
(6.5)**:** G_1 and G_2 both occur within the scope of a single negation before and after the transformation.
(6.6)**:** G_1 and G_2 both occur within the scope of a single negation before and after the transformation.
(6.7)**:** G and H both occur within the scope of a single negation before the transformation, afterward G occurs within 2 negations and H occurs within 1.
(6.14)**:** G_1 occurs within a two negations before the transformation and one afterward. No other subformulas occur within negation prior to the transformation.
(6.20)**:** H_1 occurs within a two negations before the transformation and one afterward. No other subformulas occur within negation prior to the transformation.

The claim then follows from the induction hypothesis,

Proposition 9
Given an incremental theory $\langle B, P[t], Q[t]\rangle$ and online progression $\langle E, F\rangle_{\geq 1}$ which are acyclic. The disjunctive logic compilation of $\langle B, P[t], Q[t]\rangle$ and $\langle E, F\rangle_{\geq 1}$ are also acyclic.
Proof.

- Select any two distinct coexisting incremental components F, G such that $F \not\prec G$ of the original incremental theory and online progression and let F', G' be the corresponding incremental components of the disjunctive logic compilation.
- As the original incremental theory and online progression are acyclic, it holds that there is no occurrence of an atom $a \in \mathit{Pred}(G) \setminus I(G)$ within F outside the scope of negation.
- Let F' and G' be the corresponding incremental components of the disjuncitve logic compilation.
- Finally, it is only necessary to observe that, by definition, $\mathit{Pred}(G') \subseteq \mathit{Pred}(G)$.
- Therefore, there is no occurrence of an atom $a \in \mathit{Pred}(G') \setminus I(G')$ within F' outside the scope of negation.
- It follows that the disjunctive logic compilation is acyclic by definition.

B.9 Proposition 10

Proposition 10
Given an online $\mathcal{BC}+$ action description $\mathcal{D}^{o\mathcal{BC}+}$, observation stream $\mathcal{O}_{n,\tilde{m}}$, and some incrementally parametrized multi-valued formula $\mathcal{Q}[t]$, the incremental theory $\langle B, P[t], Q[t]\rangle^{\mathcal{D}^{o\mathcal{BC}+},\mathcal{Q}[t]}$ and online progression $\langle E, F\rangle_{\geq 1}^{\mathcal{O}_{n,\tilde{m}}}$ is modular and mutually revisable.

Proof.

Given a multi-valued signature σ, by $At_{int}(\sigma)$ we define the set of atoms $c = v$ such that $c \in \sigma$ and $v \in Dom(c)$ if c is an internal constant or $v = \mathtt{u}$ otherwise.

Furthermore, given a module $\mathbb{G} = \langle G, I, O\rangle$ by $F(\mathbb{G})$ we denote G.

Modular:

By definition of $\langle B, P[t], Q[t]\rangle^{\mathcal{D},\mathcal{Q}[t]}$ and $\langle E, F\rangle_{n}^{\mathcal{O}_{n,\tilde{m}}}$ we have the following:

1. $Pred(B) \subseteq At(0{:}\sigma_{\mathrm{F}})$;
2. $Head(B) \subseteq At_{int}(0{:}\sigma_{\mathrm{F}})$;
3. $Pred(P[t/i]) \subseteq At((i{-}1){:}\sigma_F \cup (i{-}1){:}\sigma_A \cup i : \sigma_F)$;
4. $Head(P[t/i]) \subseteq At_{int}(i{:}\sigma_{\mathrm{F}} \cup (i{-}1){:}\sigma_{\mathrm{A}})$;
5. $Pred(Q[t/k]) \subseteq At(0{:}\sigma_F) \cup \bigcup_{1\leq j\leq k} At(j{:}\sigma_F \cup j{:}\sigma_A)$;
6. $Head(Q[t/k]) = \emptyset$;
7. $Pred(E_i[e_i]) \subseteq At(0{:}\sigma_F) \cup \bigcup_{1\leq j\leq e_i} At(j{:}\sigma_F \cup j{:}\sigma_A)$;
8. $Head(E_i[e_i]) \subseteq At_u(0{:}\sigma_F) \cup \bigcup_{1\leq j\leq e_i} At_u(j{:}\sigma_{\mathrm{EF}} \cup j{:}\sigma_{\mathrm{EA}})$;
9. $(Head(E_i[e_i]) \cap Head(E_j[e_j])) = \emptyset$ for $i \neq j$; and
10. $Pred(F_i[f_i]) = Head(F_i[f_i]) = \emptyset$.

We show that, for any $k \geq 0$, $\mathbb{P}_k$ is defined such that

$$Pred(F(\mathbb{P}_k)) \subseteq At(0{:}\sigma_{\mathrm{F}}) \bigcup_{1\leq i\leq k} At(i{:}\sigma_{\mathrm{F}} \cup (i{-}1){:}\sigma_{\mathrm{A}}), \text{ and}$$
$$Head(F(\mathbb{P}_k)), O(\mathbb{P}_k) \subseteq At_{int}(0{:}\sigma_{\mathrm{F}}) \bigcup_{1\leq i\leq k} At_{int}(i{:}\sigma_{\mathrm{F}} \cup (i{-}1){:}\sigma_{\mathrm{A}})$$

by induction.

base

$$\mathbb{P}_0 = \langle Simple(B, I(B)), I(B), Pred(Simple(B, I(B))) \setminus I(B)\rangle$$

is trivially defined. In addition:

$$\begin{aligned} \mathit{Pred}(F(\mathbb{P}_0)) &\subseteq \mathit{At}(0{:}\sigma_{\mathrm{F}}), \text{ and} \\ \mathit{Head}(F(\mathbb{P}_0)) &\subseteq \mathit{At}_{\mathit{int}}(0{:}\sigma_{\mathrm{F}}) \end{aligned}$$

from 1 and 2 above. It follows by the definition of *Simple* that

$$O(F(\mathbb{P}_0)) = \mathit{Head}(F(\mathbb{P}_k)).$$

inductive ($i \geq 1$)

- Assume that $\mathbb{P}_{i-1}$ is defined such that

$$\begin{aligned} \mathit{Pred}(F(\mathbb{P}_{i-1})) &\subseteq \mathit{At}(0{:}\sigma_{\mathrm{F}}) \bigcup_{1 \leq j \leq i-1} \mathit{At}(j{:}\sigma_{\mathrm{F}} \cup (j-1){:}\sigma_{\mathrm{A}}), \text{ and} \\ \mathit{Head}(F(\mathbb{P}_{i-1})), O(\mathbb{P}_{i-1}) &\subseteq \mathit{At}_{\mathit{int}}(0{:}\sigma_{\mathrm{F}}) \bigcup_{1 \leq j \leq i-1} \mathit{At}_{\mathit{int}}(j{:}\sigma_{\mathrm{F}} \cup (j-1){:}\sigma_{\mathrm{A}}) \end{aligned}$$

- Let $X = O(\mathbb{P}_{i-1}) \cup I(P[t/i])$. It holds that

$$\langle \mathit{Simple}(P[t/i], X), X, \mathit{Pred}(\mathit{Simple}(P[t/i], X)) \setminus X \rangle \tag{B.20}$$

  is trivially defined.
- By definition, $O(\mathbb{P}_{i-1})$ and $O((\text{B.20}))$ are disjoint.
- Furthermore, by the induction hypothesis it is clear that $F(\mathbb{P}_{i-1})$ is negative on $O((\text{B.20}))$.
- Similarly, by 4 and the induction hypothesis it holds that $P[t/i]$ (and therefore $\mathit{Simple}(P[t/i], X)$) is negative on $O(\mathbb{P}_{i-1})$.
- Finally, it holds that there are no loops in

$$\mathrm{DG}[F(\mathbb{P}_{i-1}) \wedge \mathit{Simple}(P[t/i], X); O(\mathbb{P}_{i-1}) \cup O((\text{B.20})]$$

  spanning $O(\mathbb{P}_{i-1})$ and $O((\text{B.20}))$:
  - Assume that there is such a loop.
  - There is then a strictly positive implication $G \rightarrow F$ in $F(\mathbb{P}_{i-1}) \wedge \mathit{Simple}(P[t/i], X)$ such that there is some $a_1 \in O((\text{B.20}))$ occurring positively in G and some $a_2 \in\in O(\mathbb{P}_{i-1})$ occurring strictly positively in F.
  - By 3 above, it holds that $O((\text{B.20})) \subseteq \mathit{At}((i-1){:}\sigma_F \cup (i-1){:}\sigma_A \cup i : \sigma_F)$.
  - It then follows from the induction hypothesis that $\mathit{Pred}(F(\mathbb{P}_{i-1})) \cap O((\text{B.20})) = \emptyset$.
  - If follows then that $G \rightarrow F$ cannot occur in $F(\mathbb{P}_{i-1})$.
  - Furthermore, by 4 above, it holds that $\mathit{Head}(\mathit{Simple}(P[t/i], X)) \cap O(\mathbb{P}_{i-1}) = \emptyset$.
  - Therefore $G \rightarrow F$ also cannot occur in $\mathit{Simple}(P[t/i], X)$.

– It then follows that there can be no such $G \rightarrow F$ in $F(\mathbb{P}_{i-1}) \wedge Simple(P[t/i], X)$.
– This contradicts our assumption that there is a loop spanning $F(\mathbb{P}_{i-1})$ and $O((\text{B.20}))$.

- It follows then that

$$\mathbb{P}_i = \mathbb{P}_{i-1} \sqcup (\text{B.20})$$

is defined.

- Furthermore, from the induction hypothesis and 3 above, it holds that

$$\begin{aligned} Pred(P_i) &= Pred(F(\mathbb{P}_{i-1})) \cup Pred(Simple(P[t/i], X)) \\ &\subseteq At(0{:}\sigma_{\mathrm{F}}) \bigcup_{1 \leq j \leq i} At(j{:}\sigma_{\mathrm{F}} \cup (j-1){:}\sigma_{\mathrm{A}}). \end{aligned}$$

- Similarly, it follows from the induction hypothesis and (4) that

$$\begin{aligned} Head(P_i) &= Head(F(\mathbb{P}_{i-1})) \cup Head(Simple(P[t/i], X)) \\ &\subseteq At_{int}(0{:}\sigma_{\mathrm{F}}) \bigcup_{1 \leq j \leq i} At_{int}(j{:}\sigma_{\mathrm{F}} \cup (j-1){:}\sigma_{\mathrm{A}}). \end{aligned}$$

- Finally, as $I(P[t/i]) = At_u((i-1){:}\sigma_{\mathrm{EA}} \cup \sigma_{\mathrm{EF}})$ it follows from the induction hypothesis that

$$\begin{aligned} O_{P_i} &= O(\mathbb{P}_{i-1}) \cup O((\text{B.20})) \\ &\subseteq At_{int}(0{:}\sigma_{\mathrm{F}}) \bigcup_{1 \leq j \leq i} At_{int}(j{:}\sigma_{\mathrm{F}} \cup (j-1){:}\sigma_{\mathrm{A}}). \end{aligned}$$

Next, we show that, for any $n \geq 0$, $\mathbb{E}_n$ is defined such that

$$\begin{aligned} Head(E_n) &= O(\mathbb{E}_n) \\ Head(E_n) &\subseteq At_u(0{:}\sigma_{\mathrm{EF}}) \bigcup_{1 \leq i \leq e_n} At_u(i{:}\sigma_{\mathrm{EF}} \cup (i-1){:}\sigma_{\mathrm{EA}}) \end{aligned}$$

by induction.

base

Trivial as

$$\mathbb{E}_0 = \langle \top, \emptyset, \emptyset \rangle.$$

inductive ($i \geq 1$)

- Assume that $\mathbb{E}_{i-1}$ is defined such that

$$\begin{aligned} Head(F(\mathbb{E}_{i-1})) &= O(\mathbb{E}_{i-1}), \text{ and} \\ Head(F(\mathbb{E}_{i-1})) &\subseteq At_u(0{:}\sigma_{\mathrm{F}}) \bigcup_{1 \leq j \leq F(\mathbb{E}_{i-1})} At_{int}(j{:}\sigma_{\mathrm{u}} \cup (j-1){:}\sigma_{\mathrm{u}}) \end{aligned}$$

- Let $X = O(\mathbb{P}_{e_i}) \cup O(\mathbb{E}_{i-1})$. It holds that

$$\langle \mathit{Simple}(E_i[e_i], X), X, \mathit{Pred}(\mathit{Simple}(E_i[e_i], X)) \setminus X \rangle \tag{B.21}$$

  is trivially defined.
- By definition, $O(\mathbb{E}_{i-1})$ and $O((\text{B.21}))$ are disjoint.
- Furthermore, by the induction hypothesis it is clear that $F(\mathbb{E}_{i-1})$ is negative on $O((\text{B.21}))$.
- Similarly, by 9 and the induction hypothesis it holds that $E[e_i]$ (and therefore $\mathit{Simple}(E_i[e_i], X)$) is negative on $O(\mathbb{E}_{i-1})$.
- Finally, it trivially holds that there are no loops in

$$\mathrm{DG}[F(\mathbb{E}_{i-1}) \wedge \mathit{Simple}(E_i[e_i], X); O(\mathbb{E}_{i-1}) \cup O((\text{B.21}))]$$

  spanning $O(\mathbb{E}_{i-1})$ and $O((\text{B.21}))$ as there are no strictly positive implications of the form $G \rightarrow F$ in $F(\mathbb{E}_{i-1}) \wedge \mathit{Simple}(E_i[e_i], X)$ and therefore no edges in the dependency are possible.
- It follows then that

$$\mathbb{E}_i = \mathbb{E}_{i-1} \sqcup (\text{B.21})$$

  is defined.
- Furthermore, as $I(E_i[e_i]) = \emptyset$ and $E_i[e_i]$ is negative on $O(\mathbb{E}_{i-1})$ (and therefore X) it holds by definition of *Simple* that

$$\mathit{Pred}(\mathit{Simple}(E_i[e_i], X)) \setminus X = \mathit{Head}(\mathit{Simple}(E_i[e_i], X)) = O((\text{B.21}))$$

- It then follows directly from the induction hypothesis that

$$\mathit{Head}(F(\mathbb{E}_{i-1}) \wedge \mathit{Simple}(E_i[e_i], X)) = O(\mathbb{E}_{i-1}) \cup O((\text{B.21})) = O(\mathbb{P}_i)$$

- Finally, from the induction hypothesis and 8 above, it holds that

$$\mathit{Head}(F(\mathbb{E}_{i-1}) \wedge \mathit{Simple}(E_i[e_i], X)) \subseteq \mathit{At}_u(0{:}\sigma_F) \cup \bigcup_{1 \leq j \leq e_i} \mathit{At}_u(j{:}\sigma_{\mathrm{EF}} \cup j{:}\sigma_{\mathrm{EA}})$$

Finally, we show that given $\mathbb{P}_k$ and $\mathbb{E}_n$ such that $k \geq \tilde{m}$ it holds that $\mathbb{R}_{n,k}$ is defined.

- It holds that

$$\mathbb{P}_k \sqcup \mathbb{E}_n \tag{B.22}$$

  is defined:

– As shown previously, it holds that

$$\begin{aligned} Head(F(\mathbb{P}_k)), O(\mathbb{P}_k) &\subseteq At_{int}(0{:}\sigma_{\mathrm{F}}) \bigcup_{1\leq j\leq i-1} At_{int}(j{:}\sigma_{\mathrm{F}} \cup (j-1){:}\sigma_{\mathrm{A}}), \\ Head(F(\mathbb{E}_n)) &= O(\mathbb{E}_n), \text{ and} \\ Head(F(\mathbb{E}_n)) &\subseteq At_u(0{:}\sigma_{\mathrm{EF}}) \bigcup_{1\leq i\leq F(\mathbb{E}_n)} At_u(i{:}\sigma_{\mathrm{EF}} \cup (i-1){:}\sigma_{\mathrm{EA}}). \end{aligned}$$

– It is then clear that $O(\mathbb{P}_k)$ and $O(\mathbb{E}_n)$ are disjoint, $F(\mathbb{P}_k)$ is negative on $O(\mathbb{E}_n)$, and $F(\mathbb{E}_n)$ is negative on $O(\mathbb{P}_k)$.

– Finally, it holds that there are no loops in

$$\mathrm{DG}[F(\mathbb{P}_k) \wedge F(\mathbb{E}_n); O(\mathbb{P}_k) \cup O(\mathbb{E}_n)]$$

spanning $O(\mathbb{P}_k)$ and $O(\mathbb{E}_n)$:

* Assume that there is such a loop.
* There is then a strictly positive implication in $G \rightarrow F$ in $F(\mathbb{P}_k) \wedge F(\mathbb{E}_n)$ such that there is some $a_1 \in O(\mathbb{P}_k)$ occurring positively in G and some $a_2 \in\in O(\mathbb{E}_n)$ occurring strictly positively in F.
* By definition, there is no strictly positive implication in $F(\mathbb{E}_n)$ which a non-empty head and body. Therefore it cannot occur in $F(\mathbb{E}_n)$.
* Furthermore, it holds that $F(\mathbb{P}_k)$ is negative on $O(\mathbb{E}_n)$.
* If follows then that $G \rightarrow F$ cannot occur in $F(\mathbb{P}_k)$.
* It then follows that there can be no such $G \rightarrow F$ in $F(\mathbb{P}_k) \wedge F(\mathbb{E}_n)$.
* This contradicts our assumption that there is a loop spanning $O(\mathbb{P}_k)$ and $O(\mathbb{E}_n)$.

– It follows that (B.22) is defined.

- Let $X = O(\mathbb{P}_k) \cup I(Q[t/k])$. It holds trivially that

$$\langle Simple(Q[t/k], X), X, Pred(Simple(Q[t/k], X)) \setminus X \rangle \tag{B.23}$$

  is defined.
- Furthermore, it follows from 6 that $Pred(Simple(Q[t/k], X)) \setminus X = \emptyset$.
- It follows trivially then that $O((\text{B.22})) \cap O((\text{B.23})) = \emptyset$, $F((\text{B.22}))$ is negative on $O((\text{B.23}))$, and there are no loops in

$$\mathrm{DG}[F((\text{B.22})) \wedge F((\text{B.23})); O((\text{B.22})) \cup O((\text{B.23}))]$$

  which span $O((\text{B.22}))$ and $O((\text{B.23}))$.
- By 6 above it is also clear that $F((\text{B.23}))$ is negative on $O((\text{B.22}))$.
- It follows then that

$$(\text{B.22}) \sqcup (\text{B.23}) \tag{B.24}$$

  is defined.

- Finally, it is only necessary to see that

$$\langle \mathit{Simple}(F_n[f_n], X_2), X_2, \mathit{Pred}(\mathit{Simple}(F_n[f_n], X_2)) \setminus X_2 \rangle,$$

  where $X_2 = O(\mathbb{P}_{f_j}) \cup O(\mathbb{E}_j) \cup I(F_j[f_j])$, is

$$\langle \top, X_2, \emptyset \rangle$$

  by definition.
- This is trivially joinable with (B.24) (and results in the same module).

It follows that $\langle B, P[t], Q[t] \rangle^{\mathcal{D}^{o\mathcal{BC}+}, \mathcal{Q}[t]}$ and $\langle E, F \rangle^{\mathcal{O}_{n,\tilde{m}}}_{\geq 0}$ is modular.

Mutually Revisable:

- Firstly, it is clear that by definition $\mathit{Pred}(F) \cap I(F) = \emptyset$ for each $F \in (5.6)$.
- Assume that there is some $F, G \in (5.6)$ such that $F \neq G$, $F \not\prec G$, and there is some $a \in (\mathit{Head}(F) \cap \mathit{Pred}(G)) \setminus I(G)$. Consider each of the following cases for (G, F):
  - $(B, P[t/i])$ $(1 \leq i \leq k)$
    By definition $\mathit{Head}(P[t/i]) \cap \mathit{Pred}(B) = \emptyset$[1].
  - $(P[t/i], P[t/i'])$ $(1 \leq i < i' \leq k)$
    By definition $\mathit{Head}(P[t/i']) \cap \mathit{Pred}(P[t/i]) = \emptyset$.
  - $(E_i[e_i], P[t/i'])$ $(1 \leq e_i < i' \leq k)$
    By definition $\mathit{Head}(P[t/i']) \cap \mathit{Pred}(E_i[e_i]) = \emptyset$.
  - $(F_j[f_j], P[t/i])$ $(1 \leq f_j < i \leq k)$
    Trivial as $\mathit{Pred}(F_j) = \emptyset$ by definition.
  - $(B, Q[t/k])$, $(P[t/i], Q[t/k])$, $(E_i, Q[t/k])$, $(F_j, Q[t/k])$
    Trivial as $\mathit{Head}(Q[t/k]) = \emptyset$ by definition.
  - $(B, E_i[e_i])$ $(1 \leq e_i \leq k)$
    By definition $\mathit{Head}(E_i[e_i]) \cap \mathit{Pred}(B_i) \subseteq 0{:}\sigma_{\mathrm{EF}} = I(B)$.
  - $(P[t/i], E_{i'}[e_{i'}])$ $(1 \leq i' \leq j)$ and $(1 \leq i \leq k)$
    By definition

$$\mathit{Head}(E_{i'}[e_{i'}]) \cap \mathit{Pred}(P[t/i]) \subseteq I(P[t/i])$$

  - $(Q[t/k], E_i[e_i])$ $(1 \leq i \leq j)$
    By definition it holds that

$$\mathit{Head}(E_i[e_i]) \subseteq \mathit{At}_u(\bigcup_{0 \leq j \leq e_i} (j{:}\sigma_{\mathrm{EF}} \cup j{:}\sigma_{\mathrm{EA}})) \qquad (e_i < \tilde{m})$$

$$\mathit{Head}(E_i[e_i]) \subseteq \mathit{At}_u(\bigcup_{0 \leq j < e_i} (j{:}\sigma_{\mathrm{EF}} \cup j{:}\sigma_{\mathrm{EA}}) \cup i{:}\sigma_{\mathrm{EF}}) \qquad \text{otherwise}$$

    As $k \geq \tilde{m}$ it follows that $\mathit{Head}(E_i[e_i]) \subseteq I(Q[t/k])$.

- $(E_i[e_i], E_{i'}[e_{i'}])$ $(1 \leq i < i' \leq j)$
  By definition there can be no two observations $(6.24)_1$ and $(6.24)_2$ such that $m_1 = m_2$ and $c_1 = c_2$, additionally no constraint may contain any external constants. Therefore it holds that $\mathit{Head}(E_{i'}[e_{i'}]) \cap (\mathit{Pred}(E_i[e_i]) = \emptyset$.
- $(B, F_j[f_j])$, $(P[t/i], F_j[f_j])$ such that $1 \leq i \leq k$, $(Q[t/k], F_j[f_j])$, and $(E_i[e_i], F_j[f_j])$ such that $1 \leq i \leq j$
  Trivial as $\mathit{Head}(F_j) = \emptyset$ by definition.

- Therefore it holds via contradiction that the incremental theory and online progression are mutually revisable.

B.10 Proposition 11

Proposition 11
Given an online $\mathcal{BC}+$ action description $\mathcal{D}^{o\mathcal{BC}+}$, observation stream $\mathcal{O}_{n,\tilde{m}}$, and some incrementally parametrized multi-valued formula $\mathcal{Q}[t]$, the incremental theory $\langle B, P[t], Q[t]\rangle^{\mathcal{D}^{o\mathcal{BC}+},\mathcal{Q}[t]}$ and online progression $\langle E, F\rangle_{\geq 1}^{\mathcal{O}_{n,\tilde{m}}}$ are acyclic.

Proof.
Given a propositional formula F and set of atoms A, by $F^{\neg\neg A}$ we denote the formula obtained from F by prepending each atom $a \in A$ with $\neg\neg$.
We consider the encoding

$$B = \begin{cases} 0{:}\mathit{Choice}(f) & \text{for each simple fluent } f \\ 0{:}G^{\neg\neg At_u(\sigma_{\mathrm{EF}})} \rightarrow 0{:}F & \text{for each static law (4.1)} \\ 0{:}\mathit{Choice}(f = \mathtt{u}) & \text{for each external fluent } f \\ 0{:}\mathit{UEC}(\sigma_F) & \end{cases}$$

$$P[t] = \begin{cases} t{:}G^{\neg\neg At_u(\sigma_{\mathrm{EF}})} \rightarrow t{:}F & \text{for each static law (4.1)} \\ (t-1){:}G^{\neg\neg At_u(\sigma_{\mathrm{EA}})} \rightarrow (t-1){:}F & \text{for each action dynamic law (4.1)} \\ (t-1){:}H^{\neg\neg X} \wedge t{:}G^{\neg\neg At(\sigma_{\mathrm{EF}})} \rightarrow t{:}F & \text{for each fluent dynamic law (4.2)} \\ t{:}\mathit{Choice}(f = \mathtt{u}) & \text{for each external fluent } f \\ (t-1){:}\mathit{Choice}(a = \mathtt{u}) & \text{for each external action } a \\ t{:}\mathit{UEC}(\sigma_F) & \\ (t-1){:}\mathit{UEC}(\sigma_A) & \end{cases}$$

$$Q[t] = \neg\neg\mathcal{Q}[t]$$

$$E_i[m_i] = \begin{cases} m{:}c = v & \text{for each observation (6.24)} \in O_i \\ \neg\neg m{:}F & \text{for each constraint (6.25)} \in O_i \end{cases}$$

$$F_i[m_i] = \top$$

[1] See the Modular part of the proof for definitions of $\mathit{Head}(F)$ and $\mathit{Pred}(F)$ for each $F \in (5.6)$.

where $X = At(\sigma_{\mathrm{F}}) \cup At_u(\sigma_{\mathrm{EA}})$.

We consider each of the incremental components in turn:

B: By definition, B precedes all other incremental components. It is therefore sufficient to verify that the atoms within $I(B)$ occur within the scope of negation. In this case, these atoms are $At_u(0{:}\sigma_{\mathrm{EF}})$, which may occur within the rules of the form

$$0{:}G^{\neg\neg At_u(\sigma_{\mathrm{EF}})} \rightarrow 0{:}F, \text{ and}$$
$$0{:}UEC(\sigma_{\mathrm{F}})$$

within the translation. Note that for the first type of rule, these input atoms cannot occur within $0{:}F$, and, while they do occur within $0{:}G^{\neg\neg At_u(\sigma_{\mathrm{EF}})}$ they only appear preceded by double negation ($\neg\neg$). Meanwhile, the UEC laws are all of the form $0{:}F \rightarrow \bot$, which can be abbreviated to $\neg 0{:}F$.

P[t/i] $(i \geq 1)$**:** By definition, $P[t/i]$ precedes $Q[t/j](j \geq 1)$, $E_j[e_j]$ $(e_j \geq i)$, and $F_j[f_j]$ $(f_j \geq i)$. Additionally, it holds that $P[t/i]$ does not coexist with any $Q[t/j]$ $(j < i)$. As such, we need only consider its relationship to:

B: As observed previously, $Pred(B) \setminus I(B) \subseteq At_{int}(0{:}\sigma_{\mathrm{F}})$. Meanwhile,

$$Pred(P[t/i]) \subseteq At((i{-}1){:}\sigma_F \cup (i{-}1){:}\sigma_A \cup i : \sigma_F).$$

These overlap only when $i = 1$. In that case, atoms within $At_{int}(0{:}\sigma_{\mathrm{F}}$ may occur in the $(t{-}1){:}H^{\neg\neg X}$ portion of the rule

$$(t{-}1){:}H^{\neg\neg X} \wedge t{:}G^{\neg\neg At(\sigma_{\mathrm{EF}})} \rightarrow t{:}F.$$

As $X \supseteq At_{int}(0 : \sigma_{\mathrm{F}})$, it follows that each such occurrence occurs within double negation by definition.

$P[t/j](j < i)$: It holds that $Pred(P[t/j] \setminus I(B) \subseteq At_{int}(i : \sigma_{\mathrm{F}} \cup (i-1) : \sigma_{\mathrm{A}})$. Meanwhile,

$$Pred(P[t/i]) \subseteq At((i{-}1){:}\sigma_F \cup (i{-}1){:}\sigma_A \cup i : \sigma_F).$$

. Similar to the the case for B, these only overlap when $j = i - 1$. In that case, their overlap is

$$(Pred(P[t/(i{-}1)]) \setminus I(P[t/(i{-}1)]) \cap Pred(P[t/i]) \subseteq At_{int}((i{-}1){:}\sigma_{\mathrm{F}},$$

which may occur in the $(t{-}1){:}H^{\neg\neg X}$ portion of the rule

$$(t{-}1){:}H^{\neg\neg X} \wedge t{:}G^{\neg\neg At(\sigma_{\mathrm{EF}})} \rightarrow t{:}F.$$

As $X \supseteq At_{int}((i{-}1){:}\sigma_{\mathrm{F}})$, it follows that each such occurrence occurs within double negation by definition.

$E_j[e_j](e_j < i)$: By definition, $E_j[e_j]$ contains no fluent atoms $k{:}f = v$ such that $k > e_j$ and no action atoms $k{:}a = v$ such that $k \geq e_j$. If $e_j < i$, these overlap only when $e_j = i - 1$. In this case, their overlap is

$$(\mathit{Pred}(E_j[e_j]) \setminus I(E_j[e_j]) \cap \mathit{Pred}(P[t/i]) \subseteq \mathit{At}((i{-}1){:}\sigma_{\mathrm{F}}.$$

The remainder follows identically to the previous case.

$F_j[f_j](f_j < i)$: Trivial.

$\mathbf{Q[t/i]}$ $(i \geq 1)$: Trivial as all occurrences of atoms within $Q[t/i]$ are within the scope of negation.

$\mathbf{E_i[e_i]}$ $(i \geq 1)$: By definition, $E_i[e_i]$ precedes $E_j[e_j](j > i)$ and $F_j[f_j](j \geq i)$. Additionally $E_i[e_i]$ does not coexist with $Q[t/j]$ $(j < e_i)$ or $F_j[f_j]$ $(j < i)$. As such, we need only consider its relationship to:

B: Observe once again that $\mathit{Pred}(B) \setminus I(B) \subseteq \mathit{At}_{int}(0{:}\sigma_{\mathrm{F}})$. These atoms may occur only within rules of the form

$$\neg\neg m{:}F.$$

The remainder is trivial as all atoms within these rules are clearly within negation.

$P[t/j]$: It holds that $\mathit{Pred}(P[t/j] \setminus I(B) \subseteq \mathit{At}_{int}(i{:}\sigma_{\mathrm{F}} \cup (i{-}1){:}\sigma_{\mathrm{A}})$. The remainder follows identically to the last case.

$E_j[e_j](j < i)$: By definition, $\mathit{Pred}(E_j[e_j]) \subseteq \mathit{At}(0{:}\sigma_F) \cup \bigcup_{1 \leq k \leq e_j} \mathit{At}(k{:}\sigma_F \cup (k{-}1){:}\sigma_A)$. Among these, the internal atoms may occur in $E_j[e_j]$ only within rules of the form

$$\neg\neg m{:}F.$$

Meanwhile, external atoms may occur in $E_j[e_j]$ only within rules of the form

$$m{:}c = v.$$

Furthermore, by definition, each such atom occurs within at most one such rule among all online components. It then follows that for any such $m{:}c = v$, if $m{:}c = v \in \mathit{Pred}(E_j[e_j])$, then $m{:}c = v \notin \mathit{Pred}(E_i[e_i])$.

$Q[t/j](j \geq e_i)$: By definition,

$$\mathit{Pred}(Q[t/i]) \setminus I(Q[t/i]) \subseteq \mathit{At}_{int}(0{:}\sigma_F) \cup \bigcup_{1 \leq j \leq i} \mathit{At}_{int}(k{:}\sigma_F \cup (k{-}1){:}\sigma_A).$$

These atoms may occur only within rules of the form

$$\neg\neg m{:}F.$$

The remainder is trivial as all atoms within these rules are clearly within negation.

$\mathbf{F_i[f_i]}$ $(i \geq 1)$: Trivial.