TU WIEN Informatics

Towards Parallel Algorithms for Abstract Dialectical Frameworks

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Logic and Computation

eingereicht von

Mathias Hofer, BSc
Matrikelnummer 01226806

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.techn. Stefan Woltran
Mitwirkung: Ass.Prof. Dipl.-Ing. Dr.techn. Johannes Wallner
Senior Lecturer Dipl.-Ing. Dipl.-Ing. Dr.techn. Wolfgang Dvořák

Wien, 30. Jänner 2022

Mathias Hofer | Stefan Woltran

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Mathias Hofer, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 30. Jänner 2022

Mathias Hofer

Danksagung

Ich möchte mich ganz besonders bei meinen Betreuern Stefan Woltran, Johannes Wallner und Wolfgang Dvořák bedanken. Diese sind mir während der Entwicklung des Systems und dem Schreiben der Arbeit mit Feedback und Ideen zur Seite gestanden.

Außerdem möchte ich mich bei meinen Eltern Lucia und Gerhard Hofer, die mir das Studium ermöglichten und mich stets unterstützten, bedanken. Sie behielten auch trotz des etwas in die Länge gezogenen Masterstudiums ihre Geduld.

Ebenfalls möchte ich meiner Schwester Helene Hofer fürs Korrekturlesen danken.

Weiters möchte ich meinen Freunden ganz herzlich dafür danken Geduld und Verständnis für das ein oder andere aufgrund der Masterarbeit abgesagte Treffen bewiesen zu haben.

Abschließend möchte ich mich bei allen bedanken, die auf irgendeine Weise zum Gelingen meiner Masterarbeit beigetragen haben.

Kurzfassung

Abstract Dialectical Frameworks (ADFs) sind eine natürliche Verallgemeinerung von Dung-style Argumentationsframeworks. Sie beschränken sich nicht auf eine Angriffsbeziehung zwischen Argumenten, sondern erlauben unter anderem auch die Definition von unterstützenden Beziehungen zwischen Argumenten. Diese Generalisierung hilft zwar in Hinblick auf die Ausdrucksstärke des Formalismus, hat jedoch negative Auswirkungen auf die Komplexität der Berechnung. Es gibt diverse Arbeiten um diese erhöhte Komplexität algorithmisch zu bändigen, bis dato jedoch kaum Parallelisierungsansätze. Das Ziel dieser Arbeit ist es diese Lücke zu schließen und zu zeigen, dass Parallelisierung einen praktikablen Weg für die Modernisierung von ADF Systemen darstellt. In Hinblick auf aktuelle und künftige Entwicklungen im Bereich der Hardware, erachten wir diesen Weg nicht nur als praktikabel, sondern auch als notwendig.

Erkenntnisse dieser Arbeit sind nicht nur theoretischer Natur, sondern resultieren in einem konkreten ADF System. Dieses System ermöglicht gängige Berechnungen diverse ADF Semantiken wahlweise in klassischer sequentieller oder aber in paralleller Manier. Das System baut auf einem algorithmischen Modell auf welches auf der Identifikation geteilter komputationaler Bausteine der verschiedenen Semantiken beruht. Diese Bausteine müssen nur einmalig implementiert werden, können dann aber von mehreren Semantiken gleichermaßen genutzt werden. Viele dieser Bausteine können außerdem mit nur wenigen Änderungen für die parallele Ausführung genutzt werden.

Um den Vorteil der Parallelisierung zu veranschaulichen werden Resultate aus Experimenten vorgestellt und diskutiert. Diese Diskussion bezieht sich auf technische Probleme und potentielle Lösungen ebendieser. Abgeschlossen wird die Arbeit mit einem Überblick über mögliche künftige Optimierungsmöglichkeiten des neuen Systems. Die bei der Implementierung gesammelte Erfahrung erlaubt außerdem noch Vorschläge für die Verbesserung von ADF Systemen im Allgemeinen.

Abstract

Abstract Dialectical Frameworks (ADFs) are a natural generalization of Dung-style argumentation frameworks. They are not restricted to the notion of attack, but also deal with supporting, dependent and redundant relations between arguments. This generalization makes ADFs more expressive at the cost of increased computational complexity. There are some algorithmic advances to tackle this increased complexity, but only little work on parallel algorithms. The goal of this thesis is to illustrate that parallelization is a viable approach to further improve ADF systems. By looking at modern multi-core CPUs, we deem parallel algorithms as necessary to fully utilize current and future hardware developments and thus making ADF systems future-proof.

The advances towards parallel algorithms result in a concrete implementation of an ADF system. This system is capable of computing common reasoning tasks for many semantics by choice either sequentially or in parallel. To keep the implementation overhead low and the system extensible for further semantics, a new algorithmic model is introduced. This model is based on shared conceptual building blocks between semantics. These building blocks only have to be implemented once and can then be used by each semantics. It was also designed with concurrent execution in mind, since many building blocks can be used by both execution models with only little changes.

The thesis also provides some experiments to illustrate the benefits of parallel execution. It also provides technical discussions and insights on problems that may occur when running things in parallel. This then concludes with an overview of possible future developments of this system to overcome these problems, but also suggestions on how ADF systems may be improved in general.

Contents

CHAPTER 1

Introduction

Formal argumentation is an important research area in artificial intelligence, with dedicated conferences, e.g. COMMA, and specialized journals, e.g. Argument and Computation. It deals with the formalization of arguments, the relation between arguments and the resolution of conflicts between arguments. It has roots in philosophy and is related to nonmonotonic logic. Dung's 1995 landmark paper [Dun95] formalized argumentation frameworks (AFs) with just the two primitive notions of argument and attack. Dung's work made argumentation respectible in mainstream AI and influenced it in a way such that it became de facto standard nowadays [Pra18, BCD07].

Abstract Dialectical Frameworks (ADFs) are a natural generalization of Dung's work. They are not restricted to the notion of attack, but also deal with supporting, dependent and redundant relations between arguments. With ADFs we gain expressiveness at the cost of increased computational complexity [DD17]. There are investigations on more tractable subclasses of ADFs [DZLW20] and algorithmic approaches based on results from such complexity analysis [LMN+18]. There is however little work on parallel algorithms for ADFs, although surveys suggest parallelization as a considerable way to improve current ADF systems [CDG+15]. There is also little work on engineering problems of ADF systems. We fill these gaps by proposing a novel parallelization framework, which we implement within a concrete ADF system as a part of TweetyProject [Thi14]. This implementation not only provides first-hand insights into engineering problems and how to tackle them, but also technical discussions on an algorithmic architecture for ADF systems.

Given some ADF D and semantics σ, our proposed system tries to find some, preferably easier to compute, partitions $\sigma_1, \ldots, \sigma_n$ s.t. $\sigma(D) = \bigcup_{i=1}^{n} \sigma_i(D)$ and $\sigma_i(D) \cap \sigma_j(D) = \emptyset$ for all $1 \leq i < j \leq n$. Intuitively, if we manage to decompose the search space of some ADF while maintaining its semantics, we are able to compute many decision problems in parallel. There is a rich family of argumentation semantics [BCG11] from which many

concepts were adopted to ADFs [BDW11, PWW13]. Since we do not have a one-size-fits-all solution to the decomposition problem that will work on arbitrary semantics out of the box, we first introduce an algorithmic model which is expressive enough for all the usual semantics, namely conflict-free, naive, admissible, preferred, complete, ground, two-valued model and stable. The idea is to represent each semantics as a pipeline of its basic building blocks, e.g. preferred semantics can be represented as the computation of a conflict-free candidate, admissibility verification and maximization. We then use this algorithmic model to first define a sequential computation framework and then discuss necessary changes in order to transform it into a parallel computation framework.

We use a reduction based approach for our system. There currently exist a handful of ADF systems, with k++adf [LMN+18], which uses reductions to SAT, being the most promising one. The additional expressiveness of Answer Set Programming (ASP) [EIK09] or Quantified Boolean Formulas (QBFs) [SBPS19] may be tempting at first, but is not exactly what we need for parallelization. If we keep the complexity of each building block within NP, and thus reducible to SAT, we gain the possibility to solve each building block in a separate thread. Hence, this rules out approaches with just a single ASP or QBF encoding.

The thesis continues with the Preliminaries chapter, which provides the reader with all necessary definitions for the upcoming chapters. The Algorithmic Model chapter then builds the foundation of the system. It uses state of the art ideas and encodings spiced up with some new algorithmic approaches. The resulting system is able to compute most of the current semantics in a traditional sequential manner. The following Parallelization chapter then builds upon this system and discusses necessary introductions, changes and also problems when dealing with parallelization. This is followed up by the Experiments chapter, which shows some interesting findings and also some technical discussions on these. There are some other existing ADF systems with different approaches, we use chapter Related Work mostly as an overview, since there is currently little related work on parallelization approaches. The thesis is then concluded with the Outlook and Conclusion chapter, which recalls some of the problems and findings but also discusses some possible paths for future improvements.

CHAPTER 2

Preliminaries

In this chapter we build a foundation for the rest of the thesis. This is done by providing basic concepts and definitions. The system we are proposing works with propositional SAT solving, hence we start with the introduction of propositional logic before introducing Abstract Dialectical Frameworks. The introduction to propositional logic is also used to clarify the notation which we use throughout the thesis.

2.1 Propositional Logic

Definition 1 *Let $\mathcal{PV}$ be a set of propositional variables, we then inductively define propositional formulas as follows:*

- *Every $p \in \mathcal{PV}$ is a formula.*
- *The truth constants $\top$ and $\bot$ are formulas.*
- *Let ϕ denote a formula, then $\neg\phi$ is also a formula.*
- *Let ϕ and ψ denote formulas, then $\phi \circ \psi$ with $\circ \in \{\wedge, \vee, \rightarrow, \leftrightarrow, \oplus\}$ is also a formula.*

Usually propositional variables are also denoted as atoms, we switch between the naming interchangeably as we see fit.

Definition 2 *An interpretation is a function $I : \mathcal{PV} \rightarrow \{\mathbf{t}, \mathbf{f}\}$ mapping propositional variables to either true or false.*

Although interpretation is defined as a function, it is often more convenient to write it as a set. Let $a, b, c \in \mathcal{PV}$ be propositional variables and I an interpretation, instead of writing $I(a) = \mathbf{t}$, $I(b) = \mathbf{f}$ and $I(c) = \mathbf{f}$ it is more concise to write $I = \{\mathbf{t}(a), \mathbf{f}(b), \mathbf{f}(c)\}$. Having I written in a set notation also comes in handy when reasoning over multiple variables like $\{\mathbf{t}(a), \mathbf{f}(b)\} \subseteq I$ or when dealing with partial interpretations. Therefore, we interchangeably switch between the function and the set notation, allowing us to write $I(a) = \mathbf{t}$ or $\mathbf{t}(a) \in I$ depending on the context. The notations $I|^a_{\mathbf{t}}$ or $I[a \mapsto \mathbf{t}]$ replace the truth assignment of a in I with $\mathbf{t}$, for instance $I = \{\mathbf{f}(a), \mathbf{f}(b)\}$ becomes $I|^a_{\mathbf{t}} = \{\mathbf{t}(a), \mathbf{f}(b)\}$.

Definition 3 *Let I denote an interpretation, we extend its definition to arbitrary formulas as follows.*

- $I(\top) = \mathbf{t}$ *and* $I(\bot) = \mathbf{f}$
- $I(\neg\phi) = \mathbf{t}$ *iff* $I(\phi) = \mathbf{f}$
- $I(\phi \wedge \psi) = \mathbf{t}$ *iff* $I(\phi) = I(\psi) = \mathbf{t}$
- $I(\phi \vee \psi) = \mathbf{t}$ *iff* $I(\phi) = \mathbf{t}$ *or* $I(\psi) = \mathbf{t}$
- $I(\phi \rightarrow \psi) = \mathbf{t}$ *iff* $I(\phi) = \mathbf{f}$ *or* $I(\psi) = \mathbf{t}$
- $I(\phi \leftrightarrow \psi) = \mathbf{t}$ *iff* $I(\phi) = I(\psi)$
- $I(\phi \oplus \psi) = \mathbf{t}$ *iff* $I(\phi) \neq I(\psi)$

Let ϕ be a propositional formula, then we write $\phi[p \mapsto \psi]$ or $\phi[p/\psi]$ to denote the formula resulting from the replacement of all occurrences of atom p in ϕ with ψ. Usually this is helpful when rewriting a formula based on a partial interpretation. Let $\phi = a \wedge b$ for instance, then $\phi[a/\bot] = \bot \wedge b$ which is logically equivalent to $\bot$. This notation can be extended to interpretations like $\phi[I] = \phi[a/\bot \mid \mathbf{f}(a) \in I][a/\top \mid \mathbf{t}(a) \in I]$. Reconsidering the previous example with $\phi = a \wedge b$ and interpretation $I = \{\mathbf{f}(a), \mathbf{t}(b)\}$ it holds $\phi[I] = \bot \wedge \top$.

Let I be an interpretation and ϕ be some formula, it holds $I \models \phi$ iff I satisfies ϕ iff $I(\phi) = \mathbf{t}$ iff ϕ is true under I. Analogously, $I \not\models \phi$ iff I does not satisfy ϕ iff $I(\phi) = \mathbf{f}$ iff ϕ is false under I. Again, we use the most convenient notation based on the context.

Definition 4 *Let φ be a propositional formula and A a (partial) interpretation called assumption. Then we define a function SAT as*

- $SAT(\varphi, A) \mapsto \begin{cases} (I, true) \text{ if } \exists I : I \models \varphi[A] \\ (\emptyset, false) \text{ otherwise} \end{cases}$
- $SAT(\varphi) \mapsto SAT(\varphi, \emptyset)$

Furthermore, let ρ denote a set of propositional formulas, then we define SAT as

- $SAT(\rho, A) \mapsto SAT(\bigwedge_{\phi \in \rho} \phi, A)$
- $SAT(\rho) \mapsto SAT(\rho, \emptyset)$

For readability we sometimes write $\neg SAT(\rho)$ as an abbreviation for $SAT(\rho) = (I, false)$ if we are not interested in the witness I. Note that it is not sufficient to just return an interpretation and omit the second value of the pair. Consider $SAT(\top) = (\emptyset, true)$, the constant $\top$ is by definition true under every interpretation I and thus also under $\emptyset$. To overcome this subtlety, the SAT function returns a pair.

The role of assumption A can be explained by the monotonicity of incremental SAT solving. Let ρ represent the SAT solver state, then it is only possible to add formulas to ρ but never remove any. An assumption only applies for one specific SAT call, but does not effect ρ for all subsequent SAT calls. This is useful, since encoding A directly into ρ would interfere with all future SAT calls on ρ.

2.2 Abstract Dialectical Framework

The definition of ADF is based on [BW10].

Definition 5 *An ADF $D = (A, L, C)$ is a tuple where*

- *A is a set of arguments*
- *$L \subseteq A \times A$ is a set of links*
- *$C = \{\varphi_a\}_{a \in A}$ is a set of propositional formulas over the parents of each argument a. φ_a is called acceptance condition of a.*

The set $par(a) = \{b \in A \mid (b, a) \in L\}$ denotes the parents of argument a. Note that sometimes the link relation L is omitted from the definition, since the acceptance condition φ_a induces the set $par(a)$. The set L can then implicitly be defined by A and C as $L = \{(b, a) \mid a \in A, b \in par(a)\}$.

Definition 6 *An interpretation is a function I mapping arguments to one of the three truth values $I : A \to \{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$. An interpretation I is two-valued if $I(a) \in \{\mathbf{t}, \mathbf{f}\}$ for all $a \in A$. An interpretation I is trivial, denoted by $I_{\mathbf{u}}$, if $I(a) = \mathbf{u}$ for all $a \in A$.*

Note that although we usually deal with three-valued interpretations when working with ADFs, we can reuse most of the notation from the propositional logic section. Let

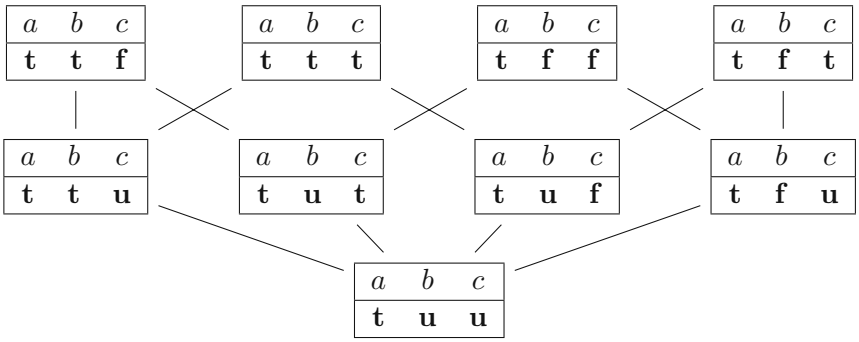


Figure 2.1: An illustration of the information ordering for three-valued interpretations.

$\varphi = a \wedge b$ denote some acceptance condition and $I = \{\mathbf{f}(a), \mathbf{u}(b)\}$ some three-valued interpretation, then $\varphi[I] = \bot \wedge b$.

Interpretation I is equally or more informative than J, denoted by $J \leq_i I$, if $J(a) \in \{\mathbf{t}, \mathbf{f}\}$ implies $J(a) = I(a)$ for all $a \in A$. We denote by $<_i$ the strict version of $\leq_i$, i.e. $J <_i I$ if $J \leq_i I$ and $\exists a \in A$ s.t. $J(a) = \mathbf{u}$ and $I(a) \in \{\mathbf{t}, \mathbf{f}\}$.

The blocks in Figure 2.1 represent interpretations and the lines between these blocks show comparability regarding $\leq_i$. The level of the block shows informativeness. The interpretation $I_1 = \{t(a), t(b), f(c)\}$ is strictly more informative than $I_2 = \{t(a), t(b), u(c)\}$ since $I_2(c) = \mathbf{u}$ but $I_1(c) = \mathbf{f}$, hence we write $I_2 <_i I_1$. Interpretation I_2 is strictly more informative than $I_3 = \{t(a), u(b), u(c)\}$, hence we write $I_3 <_i I_2$. Since $<_i$ is transitive we also have $I_3 <_i I_1$. This information ordering is especially useful if we are looking for maximal interpretations. Take another interpretation $I_4 = \{t(a), t(b), t(c)\}$, then it holds $I_1 \not\leq_i I_4$ and $I_4 \not\leq_i I_1$ which implies that in general we can have more than one maximal interpretation.

Definition 7 *A semantics σ relative to ADF D, denoted as $\sigma(D)$, is a selection of interpretations of D, i.e. $\sigma(D) \subseteq \{\mathbf{t}, \mathbf{f}, \mathbf{u}\}^A$.*

A semantics selects the interpretations of some ADF based on some common properties. Consider a maximality property regarding the information ordering $<_i$ for instance. Recall Figure 2.1, a semantics defined by this single maximality property would only contain interpretations of the top level and thus only two-valued interpretations. Later we provide definitions for the most common semantics. There are various definitions for some of them throughout the literature, some of them are equivalent but some are conflicting. The following definitions are the ones used and implemented by the proposed system.

Definition 8 *Let $D = (A, L, C)$ be an ADF and σ some semantics, we define the following reasoning tasks*

- *Cred*$_\sigma$*: Given* $a \in A$*, is there an* $I \in \sigma(D)$ *s.t.* $I(a) = \mathbf{t}$*?*
- *Skept*$_\sigma$*: Given* $a \in A$*, is* $I(a) = \mathbf{t}$ *for all* $I \in \sigma(D)$*?*
- *Exists*$_\sigma$*: Is there an* $I \in \sigma(D)$ *with* $I \neq I_u$*?*
- *Ver*$_\sigma$*: Given an interpretation* I*, is* $I \in \sigma(D)$*?*

There are further reasoning tasks, besides the computation of all interpretations relative to some semantics σ. The proposed system is capable of computing many of them in parallel, as we will see in a later chapter.

Definition 9 *Let* $D = (A, L, C)$ *be an ADF and* I *a three valued interpretation. Interpretation* I *is confict-free in* D *iff for all* $a \in A$*,*

- $I(a) = \mathbf{t}$ *implies* $\varphi_a[I]$ *is satisfiable, and*
- $I(a) = \mathbf{f}$ *implies that* $\varphi_a[I]$ *is refutable.*

An interpretation I *is naive in* D *iff* I *is* $\leq_i$*-maximal conflict-free in* D*.*

The conflict-free semantics is probably the most important one, since it also defines basic properties for many other semantics. This becomes obvious for the naive semantics by looking at its definition, but also the admissible and therefore the preferred and complete semantics are a subset of the conflict-free semantics. Intuitively, an interpretation I is conflict free if the truth value of an argument under I is consistent with the satisfiability respectively refutability of its acceptance condition under I.

Definition 10 *Let* $D = (A, L, C)$ *be an ADF and* I *a two-valued interpretation. Interpretation* I *is a model in* D *iff for all* $a \in A$ *it holds* $I(a) = \mathbf{t}$ *iff* $\varphi_a[I]$ *is satisfiable.*

The two-valued model semantics can be seen as a two-valued version of the conflict-free semantics. It also links the truth value of an argument under an interpretation I with the satisfiability of its acceptance condition under I.

Definition 11 *Let* $D = (A, L, C)$ *be an ADF and* I *a three-valued interpretation over* D*. Define the characteristic operator* $\Gamma_D(I) = J$ *as*

$$J(a) = \begin{cases} \mathbf{t} \text{ if } \varphi_a[I] \text{ is a tautology} \\ \mathbf{f} \text{ if } \varphi_a[I] \text{ is unsatisfiable} \\ \mathbf{u} \text{ otherwise} \end{cases}$$

The characteristic operator determines the truth assignment of some argument a based on all possible two-valued evaluations of $\varphi_a[I]$. This implicitly happens by checking for tautology respectively unsatisfiability. For a more detailed discussion we refer to the Approximation Fixpoint Theory section in [BES+17].

Definition 12 *Let $D = (A, L, C)$ be an ADF and I a three-valued interpretation.*

- *I is complete in D iff $I = \Gamma_D(I)$.*
- *I is admissible in D iff $I \leq_i \Gamma_D(I)$.*
- *I is preferred in D iff I is $\leq_i$-maximal admissible in D.*
- *I is grounded in D iff I is the $\leq_i$-least fixpoint of Γ_D.*

Note that based on these definitions one can observe subset relations. Clearly, the preferred semantics is a subset of the admissible semantics. It also holds $I = \Gamma_D(I)$ implies $I \leq_i \Gamma_D(I)$, hence the complete semantics is a subset of the admissible semantics. Furthermore, since the grounded interpretation is the $\leq_i$-least fixpoint, it clearly is within the set defined by $I = \Gamma_D(I)$, hence is also complete.

Definition 13 *Let $D = (A, L, C)$ be an ADF and I be a two-valued model of D. Define the reduced ADF D^I with $D^I = (A^I, L^I, C^I)$, where*

- $A^I = \{a \in A \mid I(a) = \mathbf{t}\}$
- $L^I = L \cap A^I \times A^I$
- $C^I = \{\varphi_a^I\}_{a \in A^I}$ *where for each $a \in A^I$, we set $\varphi_a^I = \varphi_a[b/\bot : I(b) = \mathbf{f}]$.*

Denote by J the unique grounded interpretation of D^I. Now the two-valued model I of D is a stable model of D iff for all $a \in A$, we find that $I(a) = \mathbf{t}$ implies $J(a) = \mathbf{t}$.

This characterisation of stable models is quite convenient when building algorithms. It is straight forward to compute if one already has algorithms for the two-valued model and the grounded semantics. All that remains to implement is the syntactical rewriting.

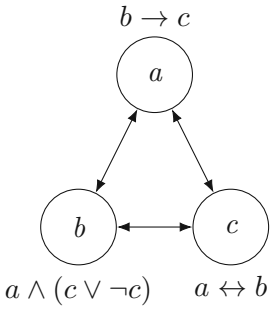
Definition 14 *Let $D = (A, L, C)$ be an ADF, then some link $(b, a) \in L$ is*

- *supporting iff for every two-valued interpretation I: $I(\varphi_a) = \mathbf{t}$ implies $I|^b_{\mathbf{t}}(\varphi_a) = \mathbf{t}$.*
- *attacking iff for every two-valued interpretation I: $I(\varphi_a) = \mathbf{f}$ implies $I|^b_{\mathbf{t}}(\varphi_a) = \mathbf{f}$.*
- *dependent iff it is neither supporting nor attacking.*

- *redundant iff it is supporting and attacking.*

A link not only defines which arguments are related, but also how they are related. In Dung's argumentation framework arguments are only related via attack, in ADFs we can additionally define the notion of support. From the definitions of supporting and attacking, the existence of links that are neither or both follows. Hence, we get two additional types, namely dependent and redundant. The supported links are denoted as $L^+ \subseteq L$, the attacking links as $L^- \subseteq L$ and the dependent links as $L^D \subseteq L$. Since L^+ and L^- also contain the redundant links, we additionally define $L^{\oplus} = L^+ \setminus L^-$ and $L^{\ominus} = L^- \setminus L^+$.

Example 1 *A more detailed version of this example can be found in [BES+17].*



L^+	L^-
(a, b)	(b, a)
(c, b)	(c, b)
(c, a)	

The table next to the ADF lists its attacking and supporting links. The link (c, b) is redundant, since it is both attacking and supporting. The links (b, c) and (a, c) are both dependent, since they are neither attacking nor supporting.

Proposition 1 *Let φ_a denote some acceptance condition and $(b, a) \in L$ be some redundant link, it then holds $\varphi_a \equiv \varphi_a[b \mapsto \top]$.*

Proof. Assume $I \models \varphi_a$, then from (b, a) being supporting it follows $I|^b_{\mathbf{t}} \models \varphi_a$ and further $I \models \varphi_a[b \mapsto \top]$. Assume $I \not\models \varphi_a$, then from (b, a) being attacking it follows $I|^b_{\mathbf{t}} \not\models \varphi_a$ and further $I \not\models \varphi_a[b \mapsto \top]$. Hence, $\varphi_a \equiv \varphi_a[b \mapsto \top]$. □

The takeaway from Proposition 1 is that we can syntactically rewrite some ADF with redundant links into a semantically equivalent one without redundant links. Hence, the more interesting link types are attacking, supporting and dependent.

A link is called bipolar if it is either attacking or supporting. An ADF is called bipolar if all of its links are bipolar. The existence of dependent links has a significant effect on the complexity of the most common reasoning tasks on ADFs. On the other hand, if an ADF is bipolar or nearly bipolar, then there is a drop in complexity as Table 2.1 shows. An ADF is called k-bipolar if it contains exactly k non-bipolar links.

	in general				bipolar / k-bipolar			
σ	$Cred_\sigma$	$Skept_\sigma$	$Exists_\sigma$	Ver_σ	$Cred_\sigma$	$Skept_\sigma$	$Exists_\sigma$	Ver_σ
cf	NP-c	trivial	NP-c	NP-c	in P	trivial	in P	in P
mod	NP-c	coNP-c	NP-c	in P	NP-c	coNP-c	NP-c	in P
stb	Σ_2^P-c	Π_2^P-c	Σ_2^P-c	coNP-c	NP-c	coNP-c	NP-c	in P
nai	NP-c	Π_2^P-c	NP-c	DP-c	in P	coNP-c	in P	in P
adm	Σ_2^P-c	trivial	Σ_2^P-c	coNP-c	NP-c	trivial	NP-c	in P
grd	coNP-c	coNP-c	coNP-c	DP-c	in P	in P	in P	in P
com	Σ_2^P-c	coNP-c	Σ_2^P-c	DP-c	NP-c	in P	NP-c	in P
prf	Σ_2^P-c	Π_3^P-c	Σ_2^P-c	Π_2^P-c	NP-c	Π_2^P-c	NP-c	coNP-c

Table 2.1: Complexity results from [DD17].

The interesting part is that bipolar and k-bipolar ADFs coincide in complexity. This results in algorithms as in [LMN+18], on which the presented system is also based. The proposed encodings are however exponential in constant k, hence the number of non-bipolar links is crucial in practice.

CHAPTER 3

Algorithmic Model

In this chapter we build a sequential system that later functions as a foundation for the parallelization approaches. This system relies on some state-of-the-art SAT encodings, some new algorithms and an algorithmic model which should help to keep the implementation effort low. The basic idea of the algorithmic model is the identification of atomic computational building blocks for each semantics. This helps to identify shared concepts between semantics. It further allows to build an algorithmic framework which is not only capable of computing the state-of-the-art semantics, but also extensible enough to capture future developments with minimal implementation overhead. Figure 3.1 shows a simplistic view of the identified building blocks and how they interact together in the computation pipeline.

The algorithm is based on incremental SAT solving, hence the pipeline starts with the creation of a solver state. This solver state represents the search space and is modified by adding clauses during the computation of interpretations. These clauses usually cut parts of the search space, for example to prevent the same interpretation from being computed again. After creation, the solver state is handed to the second block of the pipeline, namely the state processor. The state processor can be used for initializations on the state and is the right building block for optimizations. It is then handed to the candidate generator, which is from now on responsible for the maintenance of the search space and the supply of interpretations for the rest of the framework. The other building blocks have no direct access to it. This is an implementation detail and is not mirrored in Algorithm 3.9, since it is not important if all the blocks are executed sequentially. If we want to compute things in parallel, it is however crucial to not expose the search space to all the building blocks, since this enables concurrent modifications, which is error prone and may result in inconsistencies. The candidate generator is also responsible for the encoding of semantic specific properties like conflict-freeness. A generated interpretation is then handed to an interpretation processor, which may modify the current candidate. This allows for additional computations before we hand

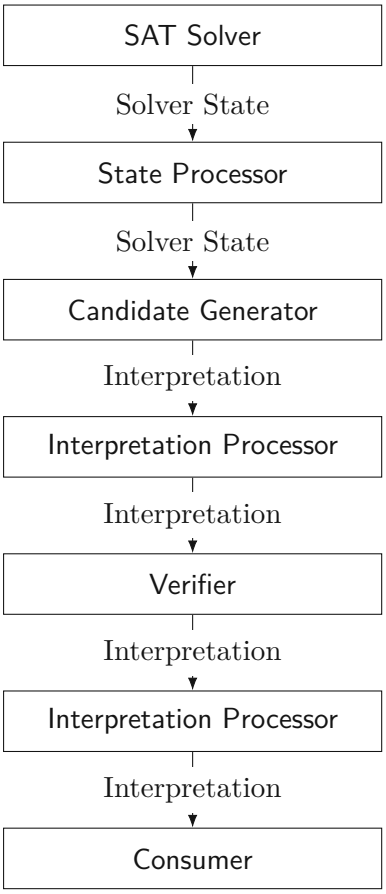


Figure 3.1: The sequential computation pipeline.

the resulting interpretation to the verifier, an example of such additional computation is maximization. In the naive semantics, such a processor takes a conflict-free interpretation and returns a maximal, i.e. naive, one. After the optional processing, the candidate is given to a verifier, which acts as a filter and dismisses all interpretations that do not pass the verification. All interpretations that pass the verification step are then handed to another optional interpretation processor. There is no conceptual difference between the first and the second interpretation processor, the mere difference is the stage at the computation pipeline.

3.1 Basic Building Blocks

Most of the ADF semantics are based on three-valued interpretations, since we base our system on SAT solving, which is two-valued, we need an extra propositional variable to represent the truth assignment of each argument. Hence, for each argument s we use the propositional variables s^t and s^f. Moreover, for each $r \in par(s)$ we introduce

propositional variables p_s^r to replace each occurrence of r in acceptance condition φ_s. These extra variables for each link are necessary to allow different assignments in different acceptance conditions if r is undecided. Consider $I(r) = \mathbf{u}$ with children $r \in par(s_1)$ and $r \in par(s_2)$, then the variables $p_{s_1}^r$ and $p_{s_2}^r$ can take different truth values. We write $\varphi_s^{\downarrow} = \varphi_s[r \mapsto p_s^r \mid (r,s) \in L]$.

Given some ADF $D = (A, L, C)$, its propositional universe is defined as $U_D = \{s^t, s^f \mid s \in A\} \cup \{p_s^r \mid (r,s) \in L\}$. Let v be a SAT witness of some formula over universe U_D, we can define a function mapping v to a three-valued interpretation of D:

$$\begin{aligned} \mathit{Extract}(v) = &\{\mathbf{u}(s) \mid v(s^t) = \mathbf{f} \text{ and } v(s^f) = \mathbf{f}\} \\ &\cup \{\mathbf{t}(s) \mid v(s^t) = \mathbf{t}\} \\ &\cup \{\mathbf{f}(s) \mid v(s^f) = \mathbf{t}\} \end{aligned}$$

Note that with two propositional variables we are able to define four truth values. Hence, it is necessary to prevent s^t and s^f from being true at the same time. This can be achieved by just adding $\bigwedge_{s \in A}(\neg s^t \vee \neg s^f)$ to the search space. In the following we always assume that this is the case.

There are a few encodings which are used by many building blocks and should therefore be discussed first. We omit correctness proofs here, since they were already introduced in [LMN+18].

$$\phi_{\neq}^I = \bigvee_{I(s)=\mathbf{t}} \neg s^t \vee \bigvee_{I(s)=\mathbf{f}} \neg s^f \vee \bigvee_{I(s)=\mathbf{u}} (s^t \vee s^f)$$

In order to prevent the same interpretation from being computed again, many candidate generators make use of formula $\phi_{\neq}^I$. Observe that $\phi_{\neq}^I$ is rendered false exactly by the propositional representation of interpretation I. Hence, only interpretations unequal to I satisfy the search space.

$$\phi_{>}^I = \bigwedge_{I(s)=\mathbf{t}} s^t \wedge \bigwedge_{I(s)=\mathbf{f}} s^f \wedge \bigvee_{I(s)=\mathbf{u}} (s^t \vee s^f)$$

Important for the maximizers, and also some verifiers, is the formula $\phi_{>}^I$. It is only satisfied by strictly more informative interpretations than I, which also coincide in the decided parts. Hence, the search space is restricted to interpretations J with $I <_i J$.

$$\phi_{\not<}^I = \bigvee_{I(s)=\mathbf{t}} s^f \vee \bigvee_{I(s)=\mathbf{f}} s^t \vee \bigvee_{I(s)=\mathbf{u}} (s^t \vee s^f)$$

The difference between $\phi_{>}^I$ and $\phi_{\not<}^I$ may look subtle at first. The formula $\phi_{\not<}^I$ rules out all interpretations which are less informative than I. It however still allows equally informative but incomparable interpretations J, hence $J \not\leq_i I$ and $I \not\leq_i J$. While $\phi_{>}^I$ is usually used to maximize some interpretation I, the formula $\phi_{\not<}^I$ is then used to rule out all smaller interpretations once a maximum was found.

The search space ρ is the last building block to discuss. In our implementation it is a set of clauses managed by some SAT solver. However, we do not want to deal with clause transformations here, therefore we define ρ as a set of arbitrary propositional formulas. By working with a set instead of one big formula, reasoning on ρ is more comfortable, since we can use set notation. Note that we can only add formulas to ρ, but never remove any formulas directly. Hence, this is also the reason why we employ some assumption based techniques later on, to make the search space appear as if some formula was removed.

3.2 Candidate Generator

The first building block to discuss is also the most important one, since it is the only mandatory building block in the computation framework, all others may be omitted. It not only provides the other building blocks with a flow of candidates to process and verify, but it also encodes basic semantics specific properties into the search space.

3.2.1 Conflict-Free

The conflict-free generator can be described with just a few steps. First make a SAT call, if the search space is satisfiable extract an interpretation from the witness, then prevent the same witness from being computed again and return the interpretation. The most important part of Algorithm 3.1 are not these recurring steps, but the encoding of the conflict-free property into the search space ρ. This has to be done only once during initialization and ensures that the resulting candidates are conflict-free.

$$\phi_s^\Gamma = (s^t \rightarrow \varphi_s^\downarrow) \wedge (s^f \rightarrow \neg\varphi_s^\downarrow)$$

The formula ϕ_s^Γ ensures that the truth value of an argument corresponds with its acceptance condition, hence if an argument is rendered true, then its acceptance condition has to be satisfied and if an argument is rendered false, its acceptance condition must not be satisfied.

$$\phi_s^{pr} = (s^t \rightarrow \bigwedge_{(s,r)\in L} p_r^s) \wedge (s^f \rightarrow \bigwedge_{(s,r)\in L} \neg p_r^s)$$

The formula ϕ_s^{pr} propagates the truth value of some argument to its children via the corresponding p_r^s variables used in their acceptance conditions.

Proposition 2 *Let $D = (A, L, C)$ be some ADF and $\rho \supseteq \bigcup_{s\in A}\{\phi_s^\Gamma, \phi_s^{pr}\}$ be some non-exhausted search space, i.e. there exists some $I = generate_{cf}(\rho)$ with $I \neq \emptyset$, then I is conflict-free in D.*

Proof. Follows from the correctness of ϕ_s^Γ and ϕ_s^{pr}. □

Algorithm 3.1: $generate_{cf}(\rho)$

```
1 (τ, sat) ← SAT(ρ)
2 if sat then
3 |   I ← Extract(τ)
4 |   ρ ← ρ ∧ φ^I_≠
5 |   return I
6 end
7 return ∅
```

Proposition 3 *Let D be some ADF and I some conflict-free interpretation of D, then $I = generate_{cf}(\rho)$ for some ρ.*

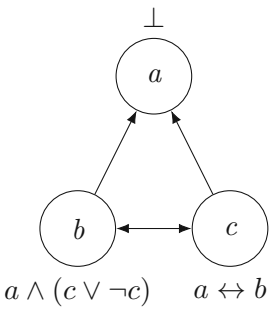
Proof. Completeness follows from the correctness of $\phi^I_{\neq}$. Assume that there is some conflict-free interpretation I which is not returned by the algorithm. Then $\phi^I_{\neq} \notin \rho$ must hold, which only happens if I was computed at some point. This is clearly a contradiction, hence I was returned for some ρ. □

3.2.2 Grounded

The grounded generator is a direct implementation of the Γ_D operator. Algorithm 3.2 starts with I_u and applies Γ_D until it computes a fixpoint. Once an argument is decided, it is fixed and not considered again in subsequent iterations. The search space ρ has to be initialized with the formulas ϕ^{Γ}_s and ϕ^{pr}_s, analogous to the conflict-free generator.

There is an important observation to make on the usage of the grounded generator in the presented enumeration framework. The generate function is called in a while loop until the search space is exhausted. Most other generators implicitly exhaust the search space by excluding recently computed candidates, but not the grounded generator. Algorithm 3.2 works differently, therefore it is necessary to make ρ unsat after the first call to ensure termination. This is done by adding $\bot$ to the search space before the return statement.

Example 2 *Consider the following ADF with arguments $\{a, b, c\}$ and acceptance conditions $\varphi_a = \bot$, $\varphi_b = a \wedge (c \vee \neg c)$ and $\varphi_c = a \leftrightarrow b$.*



Algorithm 3.2: $generate_{grd}(\rho)$

```
1  I_new ← I_u
2  I_old ← I_u
3  repeat
4  |  I_old ← I_new
5  |  ρ ← ρ ∪ {s^t | I_old(s) = t} ∪ {s^f | I_old(s) = f}
6  |  for s ∈ A with I_old(s) = u do
7  |  |  if ¬SAT(ρ ∪ {¬φ_s^↓}) then
8  |  |  |  I_new(s) ↦ t
9  |  |  else if ¬SAT(ρ ∪ {φ_s^↓}) then
10 |  |  |  I_new(s) ↦ f
11 |  |  else
12 |  |  |  I_new(s) ↦ u
13 |  |  end
14 |  end
15 until I_new = I_old
16 ρ ← ρ ∪ {⊥}
17 return I_new
```

Further assume $\rho = \{\phi_a^\Gamma, \phi_a^{pr}, \phi_b^\Gamma, \phi_b^{pr}, \phi_c^\Gamma, \phi_c^{pr}\}$ *holds. Algorithm 3.2 begins with* $I_{new} = I_{old} = I_u$*, therefore no decided arguments can be fixed in* ρ*.*

The inner for-loop begins the iteration with argument a *for which the first tautology check fails, i.e.* $\neg SAT(\rho \cup \{\top\})$ *does not hold, since* ρ *is initially clearly sat which does not change by adding* $\top$*. The algorithm continues with the successful unsat check* $\neg SAT(\rho \cup \{\bot\})$*, fixing* $I_{new}(a) = \mathbf{f}$*. The next arguments to consider are* b *and* c*, the acceptance conditions of both are currently clearly satisfiable and refutable, therefore the checks fail and they remain undecided. At the end of the for-loop it now holds* $I_{new} = \{\mathbf{f}(a), \mathbf{u}(b), \mathbf{u}(c)\}$*.*

Since $I_{new} \neq I_{old}$ *holds, the algorithm continues by setting* $I_{old} = I_{new}$ *and adding* a^f *to* ρ*. Hence, by fixing* $I(a) = \mathbf{t}$ *the remaining acceptance conditions logically render to* $\varphi_b = \bot$ *and* $\varphi_c = \top$*. Since* a *is decided, the for-loop now only iterates over* b *and* c*. Starting with argument* b*, the first check* $\neg SAT(\rho \cup \{\top\})$ *fails again, the second check* $\neg SAT(\rho \cup \{\bot\})$ *however holds, deciding* $I(b) = \mathbf{f}$*. The loop continues with argument* c *for which the first check* $\neg SAT(\rho \cup \{\bot\})$ *succeeds, deciding* $I(c) = \mathbf{t}$*. The for-loop finished with* $I_{new} = \{\mathbf{f}(a), \mathbf{f}(b), \mathbf{t}(c)\}$*.*

Since $I_{new} \neq I_{old}$ *holds, the algorithm continues by setting* $I_{old} = I_{new}$ *and adding* b^f *and* c^t *to* ρ*. There are no remaining undecided arguments, the inner for-loop is therefore skipped leaving* I_{new} *untouched. Now* $I_{old} = I_{new}$ *holds, which terminates the outer loop.*

The algorithm now renders ρ *unsat by adding* $\bot$*, this indicates the framework that the search space is exhausted. Then the found grounded interpretation* $\{\mathbf{f}(a), \mathbf{f}(b), \mathbf{t}(c)\}$ *is*

returned.

Proposition 4 *Algorithm 3.2 always terminates.*

Proof. Observe that the inner for-loop only iterates over undecided arguments, furthermore all the decided arguments are fixed and never considered again. Therefore, either the inner-for loop decides another argument or $I_{new} = I_{old}$ and the algorithm terminates. Since the number of undecided arguments is finite and monotonically decreasing, at one point it is not possible to decide another argument, which again results in the case $I_{new} = I_{old}$. Hence, the algorithm always terminates. □

Proposition 5 *Let $D = (A, L, C)$ be some ADF and $\rho = \bigcup_{s \in A}\{\phi_s^\Gamma, \phi_s^{pr}\}$, then $I = generate_{grd}(\rho)$ is the grounded interpretation of D.*

Proof. Observe that the inner for-loop computes one application of the Γ_D operator, while the outer while-loop applies Γ_D until a fixpoint is reached. Hence, by construction the algorithm computes $\Gamma_D(\ldots\Gamma_D(I_u))$, which is the $\leq_i$-least fixpoint of Γ_D and therefore the grounded interpretation of D. □

3.2.3 Two-Valued Model

The Algorithm 3.3 for two-valued models coincides with the conflict-free generator. The key difference is the semantical encoding.

$$\phi_s^{mod} = (s^t \leftrightarrow \varphi_s) \wedge (\neg s^t \rightarrow s^f)$$

In principle this encoding is similar to the conflict-free encoding, but it is possible to take shortcuts, since only two-valued interpretations have to be computed. Hence, the undecided case can be skipped entirely and therefore the extra link variables, which allows the usage of φ_s directly. There is also no need for the formula ϕ_s^{pr}. It remains an encoding which renders s^t to true if the acceptance condition is satisfied, otherwise s^f is set to true. Note that this encoding still uses two propositional variables for each argument to make use of the *Extract* function.

Proposition 6 *Let $D = (A, L, C)$ be some ADF and $\rho \supseteq \bigcup_{s \in A}\{\phi_s^{mod}\}$ be some non-exhausted search space, i.e. there exists some $I = generate_{mod}(\rho)$ with $I \neq \emptyset$, then I is a two-valued model in D.*

Proof. We show that ϕ_s^{mod} is correct, from which the proposition then follows. Observe that $s^t \leftrightarrow \varphi_s$ ensures $I(s) = \mathbf{t}$ iff $\varphi_s[I]$ is satisfiable. If s^t is not rendered true then $\neg s^t \rightarrow s^f$ forces s^f to be true. Hence, exactly one of s^t and s^f is true, which ensures that interpretations are two-valued. Therefore, ϕ_s^{mod} encodes the properties of two-valued models. □

Algorithm 3.3: $generate_{mod}(\rho)$

```
(τ, sat) ← SAT(ρ)
if sat then
    I ← Extract(τ)
    ρ ← ρ ∧ φ^I_≠
    return I
end
return ∅
```

Proposition 7 *Let D be some ADF and I some two-valued model of D, then $I = generate_{mod}(\rho)$ for some ρ.*

Proof. The argumentation is analogous to the conflict-free algorithm. □

3.3 State Processor

The state processor is the place to put semantics independent encodings. These are added to the search space before it is handed to the candidate generator. This makes mostly sense for optimizations.

3.3.1 K-Bipolar Optimization

The k-bipolar encodings are a main result of [LMN+18] and an important optimization that is used by our system.

$$\phi_r^{bip} = \left(r^t \rightarrow \left(\bigwedge_{(s,r)\in L_D^{\ominus}} (\neg s^f \rightarrow p_r^s) \wedge \bigwedge_{(s,r)\in L_D^{\oplus}} (\neg s^t \rightarrow \neg p_r^s) \right) \right) \wedge \left(r^f \rightarrow \left(\bigwedge_{(s,r)\in L_D^{\ominus}} (\neg s^t \rightarrow \neg p_r^s) \wedge \bigwedge_{(s,r)\in L_D^{\oplus}} (\neg s^f \rightarrow p_r^s) \right) \right)$$

Consider some interpretation I with $I(s) = \mathbf{u}$, the implication $\neg s^f \rightarrow p_r^s$ then forces p_r^s to be true for all $(s, r) \in L^{\ominus}$. This renders the acceptance conditions of arguments r false by definition of attacking. If we had $I(r) = \mathbf{t}$ for some child r, then clearly $I \leq_i \Gamma_D(I)$ does not hold. Hence, ϕ_r^{bip} acts as an admissibility check by setting the link variables p_r^s for undecided parents according to the polarity of the link.

$$\phi_s^{\Gamma_?} = \left(s^t \rightarrow \left(\bigwedge_{I_X \in \mathcal{V}(X_s)} \left(\varphi_s^{\downarrow, I_X} \vee \bigvee_{I_X(r)=\mathbf{t}} r^f \vee \bigvee_{I_X(r)=\mathbf{f}} r^t \right) \right) \right) \wedge \left(s^f \rightarrow \left(\bigwedge_{I_X \in \mathcal{V}(X_s)} \left(\neg\varphi_s^{\downarrow, I_X} \vee \bigvee_{I_X(r)=\mathbf{t}} r^f \vee \bigvee_{I_X(r)=\mathbf{f}} r^t \right) \right) \right)$$

The set $X_s = \{r \mid \exists (r, s) \in L_D^?\}$ contains the parents on which s depends, furthermore $\mathcal{V}(X_s)$ denotes the set of two-valued interpretations over X_s. Assume r^t to be true, then the right side of the implication must be satisfied. First observe that the check on the right side is active for all comparable completions $J \leq_i I_X$. It then basically checks if there is one completion under which $\varphi_s^{\downarrow, I_X}$ does not hold anymore, in which case s^t cannot be fixed to true. Note that this is again in line with the concept of admissibility, once an argument is decided it is not switched by fixing some undecided argument.

Hence, by adding ϕ_r^{bip} and $\phi_s^{\Gamma_?}$ to the search space, we can omit an extra admissibility check.

Algorithm 3.4: $process_{kbip}(\rho)$

```
for s ∈ A do
    ρ ← ρ ∪ {φ_s^bip, φ_s^Γ?}
end
return ρ
```

Let $D = (A, L, C)$ be an ADF, Algorithm 3.4 then processes the search space ρ by adding the just discussed bipolarity optimizations to it. Note that it is not necessary to check if D has dependent links in order to add $\phi_s^{\Gamma_?}$, if D is bipolar the set X_s is empty and the encoding therefore by definition omitted.

3.4 Verifier

The main task of a verifier is to ensure certain properties and to act as a filter if those are not met. A verifier is needed if a property cannot be directly encoded into the search space, at least not under complexity considerations. In such cases extra checks can be performed via a verifier. If a verifier returns false for an interpretation, this interpretation is then discarded from the pipeline.

3.4.1 Complete

An interpretation is considered complete if it is not possible to decide any further undecided arguments. On the other hand, if it is possible to decide another argument, then the interpretation was not complete. This describes how the complete verifier works,

if it is possible to decide another argument, it returns false. If none of the currently undecided arguments can be decided, then it returns true.

Algorithm 3.5: $verify_{com}(I)$

```
1 φ ← ⋀_{s∈A}(φ_s^Γ ∧ φ_s^pr)
2 ρ ← ρ ∧ ⋀_{I(s)=t} s^t ∧ ⋀_{I(s)=f} s^f
3 for s ∈ A with I_old(s) = u do
4 |   if ¬SAT(ρ ∧ ¬φ_s^↓) then
5 |   |   return false
6 |   else if ¬SAT(ρ ∧ φ_s^↓) then
7 |   |   return false
8 end
9 return true
```

It is important to note that Algorithm 3.5 requires the given interpretation to be admissible. This makes it possible to fix the already decided arguments and only check the undecided ones.

Proposition 8 *Let D be some ADF and I be some admissible interpretation, then $verify_{com}(I) = true$ iff I is complete in D.*

Proof. Assume $verify_{com}(I) = true$ holds. Observe that the for-loop implements the partial application of the characteristic operator Γ_D on I. Since I is admissible, it is sufficient to only consider the undecided arguments in the for-loop. By assumption, neither the tautology nor the unsatisfiability checks hold for any argument. Hence, it holds $I = \Gamma_D(I)$ and therefore I is complete.

Assume I is complete in D. Hence, $I = \Gamma_D(I)$ holds, which means that the tautology and unsatisfiability checks for the undecided arguments fail. Therefore, the for-loop is not exited by returning false, from which $verify_{com}(I) = true$ follows. □

3.4.2 Stable

The stable verifier is a direct implementation of Definition 13. Algorithm 3.6 computes the reduct of ADF $D = (A, L, C)$ and two-valued interpretation I, denoted as D^I. It then computes the grounded interpretation $grd(D^I)$ and checks if it is equal to I, in which case true is returned.

Algorithm 3.6: $verify_{stb}(I)$

```
1 I_grd ← generate_grd(D^I)
2 return I_grd = I
```

Proposition 9 *Let D be some ADF and I be some two-valued interpretation, then $verify_{stb}(I) = true$ iff I is stable in D.*

Proof. Observe that the algorithm is a direct implementation of the definition. □

3.5 Interpretation Processor

An interpretation processor is the place to further modify a candidate. This can be done at two stages of the pipeline, depending on which is more convenient. An interpretation processor also has the possibility to return clauses which are then added to the search space. The pipeline does not enforce any restrictions on the interpretation processors, hence they do not have to modify any interpretations or return any clauses.

3.5.1 Conflict-Free Maximizer

In order to compute naive interpretations, the conflict-free maximizer is needed. Algorithm 3.7 expects a conflict-free interpretation and looks for more informative interpretations according to $<_i$ by adding $\phi^I_>$ to the maximization state until it becomes unsat. The unsat state then signals that there is no larger conflict-free interpretation, hence the last one found is maximal and can be returned.

A fresh solver state is used for each call instead of an assumption based technique because of two reasons. First, since usually more than one $\phi^I_>$ formula is added to the state for each call, the state becomes polluted faster in comparison to the verifiers defined in the next chapter. Second, since the processors do not have global state, it is easier to scale for the parallel framework. There is no need for some external synchronization mechanism and a new processing task can be spawned for each new candidate. This is especially important since an interpretation processor communicates updates back to the search space. Ideally, less redundant interpretations are computed.

Algorithm 3.7: $maximize_{cf}(I)$

```
1 φ ← φ^I_> ∧ ⋀_{s∈A}(φ^Γ_s ∧ φ^pr_s)
2 I_max ← I
3 (w, sat) ← SAT(φ)
4 while sat do
5 |   I_max ← Extract(w)
6 |   φ ← φ ∧ φ^I_>
7 |   (w, sat) ← SAT(φ)
8 end
9 return (I_max, φ^{I_max}_≮)
```

Proposition 10 *Let D be some ADF and I be some conflict-free interpretation, then interpretation J with $maximize_{cf}(I) = (J, \phi^J_{\not<})$ is maximally conflict-free in D.*

Proof. The interpretation J is conflict-free because of the formulas ϕ_s^{Γ} and ϕ_s^{pr}, or by assumption if I is already maximal, since then $I = J$. Assume that J is not maximal, then φ is satisfiable since there is some interpretation $J' >_i J$. It then follows that some interpretation $J'' \geq_i J'$ is returned, which clearly contradicts $maximize_{cf}(I) = (J, \phi_{\not<}^{J})$. Hence, if J is returned it has to be maximal. □

3.5.2 Admissible Maximizer

Although Algorithm 3.8 is similar to the conflict-free maximizer, only admissible interpretations must be considered as more informative according to $<_i$. Hence, again the formulas ϕ_s^{bip} and $\phi_s^{\Gamma_?}$ are used to ensure the computation of admissible interpretations.

Algorithm 3.8: $maximize_{adm}(I)$

```
1 φ ← φ^I_> ∧ ⋀_{s∈A}(φ^Γ_s ∧ φ^pr_s ∧ φ^{Γ?}_s ∧ φ^bip_s)
2 I_max ← I
3 (w, sat) ← SAT(φ)
4 while sat do
5 |   I_max ← Extract(w)
6 |   φ ← φ ∧ φ^I_>
7 |   (w, sat) ← SAT(φ)
8 end
9 return (I_max, φ^{I_max}_{⊀})
```

Proposition 11 *Let D be some ADF and I be some admissible interpretation, then interpretation J with $maximize_{adm}(I) = (J, \phi_{\not<}^{J})$ is maximally admissible in D.*

Proof. Analogous to the conflict-free maximizer. □

3.6 Putting Things Together

An algorithmic view of the computation pipeline is given by Algorithm 3.9. It illustrates the control and data-flow between the building blocks. The algorithm enumerates all interpretations of a given ADF D. The computation starts by preparing the search space ρ, which is done by the state processor and then implicitly by the candidate generator, represented by *processState* and *generate*. Then it exhaustively computes candidates, processes, verifies and returns them. The interpretation processors at the two different stages are denoted by *processUnverified* and *processVerified*. They return a tuple consisting of the processed candidate I and a propositional formula which is added to ρ. This formula is usually used to cut off parts of the search space after the interpretation processing is done.

Algorithm 3.9: enumerate(D)

```
1  I ← ∅
2  ρ ← ∅
3  ρ ← processState(ρ)
4  I ← generate(ρ)
5  while I ≠ ∅ do
6  |  (I, ψ) ← processUnverified(I)
7  |  ρ ← ρ ∪ {ψ}
8  |  if verify(I) then
9  |  |  (I, ψ) ← processVerified(I)
10 |  |  ρ ← ρ ∪ {ψ}
11 |  |  I ← I ∪ {I}
12 |  end
13 |  I ← generate(ρ)
14 end
15 return I
```

Not all semantics need every building block, the algorithm can therefore be seen as a schema that can be instantiated with only the necessary building blocks. If a building block is not used, the following defaults are assumed.

- $processState(\sigma) \mapsto \sigma$
- $processUnverified(I) \mapsto I$
- $verify(I) \mapsto true$
- $processUnverified(I) \mapsto I$

Most of the defaults are just the identity function, except for the verifier. If there is no need for verification, then by returning always true the framework does not filter any candidates. Since none of these defaults interfere with the results of the following algorithms, the unused building blocks are omitted completely for readability. However, there only exists the implementation of Algorithm 3.9, all of the following algorithms can be seen as configurations of Algorithm 3.9, simplified for readability.

3.6.1 Conflict-Free

It is possible to compute conflict-free interpretations directly with just a generator, hence there is no need for further building blocks. Only the exhaustive computation of interpretations in a loop remains from the framework. Since there usually is a vast number of conflict-free interpretations for an ADF, it rarely is a good idea to exhaustively enumerate them all. However, Algorithm 3.10 is a byproduct of more advanced semantics, since a conflict-free generator is the foundation for many of them.

Algorithm 3.10: $enumerate_{cf}(D)$

```
1 I ← ∅
2 ρ ← ⋃_{s∈A} {φ_s^Γ, φ_s^pr}
3 I ← generate_cf(ρ)
4 while I ≠ ∅ do
5 |   I ← I ∪ {I}
6 |   I ← generate_cf(ρ)
7 end
8 return I
```

3.6.2 Naive

To compute naive interpretations it is necessary to initialize the framework with a second building block, namely the conflict-free maximizer. Since there is no verifier involved, it is not relevant on which stage of the pipeline the maximization is performed. Algorithm 3.11 again exhaustively computes conflict-free interpretations I in a while loop, but additionally calls $maximize_{cf}(I)$. Recall that it is not necessary to compute non-maximal interpretations $I \leq_i I_{nai}$ in subsequent loop iterations, hence these interpretations are cut off from the search space via the formula $\phi_{\not\leq}^{I_{nai}}$.

Algorithm 3.11: $enumerate_{nai}(D)$

```
1  I ← ∅
2  ρ ← ⋃_{s∈A} {φ_s^Γ, φ_s^pr}
3  I ← generate_cf(ρ)
4  while I ≠ ∅ do
5  |   (I_nai, φ_{≰}^{I_nai}) ← maximize_cf(I)
6  |   ρ ← ρ ∧ φ_{≰}^{I_nai}
7  |   I ← I ∪ {I_nai}
8  |   I ← generate_cf(ρ)
9  end
10 return I
```

3.6.3 Two-Valued Model

It is again possible to compute two-valued models by just using a generator, as illustrated by Algorithm 3.12. As a rule of thumb, one can look at the complexity of a certain semantics to determine the number of building blocks needed for its computation. Since the two-valued model semantics is comparably tame, as is the conflict-free semantics, there is no need for further building blocks.

Algorithm 3.12: $enumerate_{mod}(D)$

```
1 I ← ∅
2 ρ ← ⋃_{s∈A}{φ_s^{mod}}
3 I ← generate_{mod}(ρ)
4 while I ≠ ∅ do
5 |   I ← I ∪ {I}
6 |   I ← generate_{mod}(ρ)
7 end
8 return I
```

3.6.4 Admissible

A second building block is needed for the computation of admissible interpretations. Conceptually one would assume a verifier, but since Algorithm 3.13 makes use of the k-bipolar encodings, the verifier can be omitted. Hence, a state processor is used to encode admissibility checks directly into the search space.

Algorithm 3.13: $enumerate_{adm}(D)$

```
1 I ← ∅
2 ρ ← ⋃_{s∈A}{φ_s^Γ, φ_s^{pr}}
3 ρ ← processState_{kbip}(ρ)
4 I ← generate_{cf}(ρ)
5 while I ≠ ∅ do
6 |   I ← I ∪ {I}
7 |   I ← generate_{cf}(ρ)
8 end
9 return I
```

The current approach outperforms alternatives in the bipolar case or in the k-bipolar case if k is sufficiently small. There are however problems if k becomes too big, especially for reasoning tasks other than exhaustive enumeration. The problem here is that a large amount of time is then spent to compute the encodings, since they are exponential in k. This may amortize itself if all interpretations are enumerated, for many reasoning tasks this is however not necessary. There are different possible ways to tackle this problem. One could try to generate k-bipolar clauses on-the-fly guided by the enumeration process. This approach is not obvious, which makes further research necessary. In the case of success it could however eliminate the start-up time spent for the expensive encodings, with currently not foreseeable impact on the enumeration performance. A simpler solution could be ADF and reasoning-task specific heuristics to determine if the k-bipolar encodings are omitted completely, which then makes it obligatory to use an admissible verifier. In order to find good heuristics, systematic experiments become necessary. We

have not investigated any of these two approaches, since the primary focus of this thesis is parallelization. It is however a possible path for future improvements of ADF systems and worthwhile to mention.

3.6.5 Preferred

Conceptually, the preferred algorithm is to the admissible algorithm what the naive algorithm is to the conflict-free algorithm. Algorithm 3.14 takes the admissible algorithm as a foundation and enriches it with a maximization step. The right building block for this task was already defined, namely the admissible maximizer. The algorithm then follows the same principle as the algorithm for the naive semantics. The formula $\phi_{\not\leq}^{I_{prf}}$ returned by the maximization step restricts the search space s.t. no interpretations $I \leq_i I_{prf}$ are computed in subsequent loop iterations.

Algorithm 3.14: $enumerate_{prf}(D)$

```
1  I ← ∅
2  ρ ← ⋃_{s∈A} {φ_s^Γ, φ_s^pr}
3  ρ ← processState_kbip(ρ)
4  I ← generate_cf(ρ)
5  while I ≠ ∅ do
6  |  (I_prf, φ_{⊀}^{I_prf}) ← maximize_adm(I)
7  |  ρ ← ρ ∪ {φ_{⊀}^{I_prf}}
8  |  I ← I ∪ {I_prf}
9  |  I ← generate_cf(ρ)
10 end
11 return I
```

Example 3 *Recall the ADF from Example 1, with arguments* $\{a, b, c\}$ *and acceptance conditions* $\varphi_a = b \rightarrow c$, $\varphi_b = a \wedge (c \vee \neg c)$ *and* $\varphi_c = a \leftrightarrow b$.

Algorithm 3.14 starts with the initialization of the seach space ρ *with the conflict-free encodings, resulting in* $\rho = \rho_{cf}$ *with* $\rho_{cf} = \{\phi_a^\Gamma, \phi_a^{pr}, \phi_b^\Gamma, \phi_b^{pr}, \phi_c^\Gamma, \phi_c^{pr}\}$. *Then the state processor applies the k-bipolar optimizations to* ρ, *which also ensures that* ρ *only returns admissible interpretations. The search space then becomes* $\rho = \rho_{cf} \cup \rho_{kbip}$ *with* $\rho_{kbip} = \{\phi_a^{bip}, \phi_a^{\Gamma_?}, \phi_b^{bip}, \phi_b^{\Gamma_?}, \phi_c^{bip}, \phi_c^{\Gamma_?}\}$.

The candidate generator then begins to exhaustively compute interpretations. Assume that the first interpretation it returns is $I = \{\mathbf{u}(a), \mathbf{u}(b), \mathbf{u}(c)\}$, *which is trivially admissible. After the addition of* $\phi_{\neq}^I$ *to* ρ *by the generator, this interpretation is then handed to the admissible maximizer. The maximizer creates a new SAT solver state that represents the search space used to find larger, w.r.t. the information ordering* $<_i$, *interpretations. The formula* $\phi_{>}^I$ *enforces at least one argument to be decided. Assume* $I(c) = \mathbf{t}$, *by*

examination of the acceptance conditions one realizes that this renders φ_a true, and by the acceptance condition φ_c it follows $I(b) = \mathbf{t}$. In fact, $I' = \{\mathbf{t}(a), \mathbf{t}(b), \mathbf{t}(c)\}$ is not only the preferred interpretation, it also is the only other admissible interpretation besides I. Hence, the maximizer finds and returns the clearly maximal interpretation I' and thus the tuple $(I', \phi^{I'}_{\not<})$. Then ρ and $\mathcal{I}$ are both updated s.t. $\phi^{I'}_{\not<} \in \rho$ and $\mathcal{I} = \{I'\}$ holds. Since $\{\phi^{I}_{\neq}, \phi^{I'}_{\not<}\} \subseteq \rho$, all the admissible interpretations are ruled out, hence $generate_{cf}(\rho) = \emptyset$ leads to the termination of the loop and the return of $\mathcal{I}$.

3.6.6 Stable

Algorithm 3.15 consists of two building blocks, each establishing one property of stable interpretations. Since each stable interpretation is a two-valued model, a two-valued model generator is used to compute candidates. A verifier then checks if a two-valued model is stable.

Algorithm 3.15: $enumerate_{stb}(D)$

```
1  I ← ∅
2  ρ ← ⋃_{s∈A} {φ_s^mod}
3  I ← generate_mod(ρ)
4  while I ≠ ∅ do
5  |   if verify_stb(I) then
6  |   |   I ← I ∪ {I}
7  |   end
8  |   I ← generate_mod(ρ)
9  end
10 return I
```

3.6.7 Grounded

Since there is only one grounded interpretation, Algorithm 3.16 skips the while loop for readability. Then only the grounded generator remains in the computation pipeline. This is again in line with the rule-of-thumb of one building block per level on the polynomial hierarchy.

Algorithm 3.16: $enumerate_{grd}(D)$

```
1  ρ ← ⋃_{s∈A} {φ_s^Γ, φ_s^pr}
2  I ← generate_grd(ρ)
3  return {I}
```

3.6.8 Complete

First observe, by a look at the definitions, that each complete interpretation is admissible. Algorithm 3.17 therefore only computes admissible candidates. This is again achieved by the addition of the k-bipolar encodings to the search space via the corresponding state processor. In combination with a conflict-free generator only admissible interpretations are computed. It then remains to filter all non-complete admissible interpretations, this is done with the complete verifier.

Algorithm 3.17: $enumerate_{com}(D)$

```
1  I ← ∅
2  ρ ← ⋃_{s∈A} {φ_s^Γ, φ_s^pr}
3  ρ ← processState_{kbip}(ρ)
4  I ← generate_{cf}(ρ)
5  while I ≠ ∅ do
6  |   if verify_{com}(I) then
7  |   |   I ← I ∪ {I}
8  |   end
9  |   I ← generate_{cf}(ρ)
10 end
11 return I
```

CHAPTER 4

Parallelization

In this section we build upon the previously defined building blocks to design a system that can run various reasoning tasks in parallel. It is important to observe that there is a difference between concurrency and parallelization. In principle, concurrency describes a structural property of a problem, in our case the semantics were decomposed into concurrent building blocks. These concurrent building blocks may be executed in parallel, but not necessarily. Parallelization therefore describes a runtime behaviour, which depends on configuration of the system and the actual hardware. Further note that also multi-threading does not equal parallel execution, since even on a single-core system one can have multiple threads managed by the operating system. This can still be beneficial, just consider a case where a thread waits for some expensive IO operation, the operating system can in the meantime schedule a different thread for execution. The takeaway here is that in reality we design a concurrent system, which may run in parallel. For our system it is not transparent when or even if certain building blocks run in parallel, but it is necessary to make sure that in either case the system works [PGB+05].

In the following, if something is executed concurrently, neither the time nor the order of execution is known. Take the stable semantics for instance, the algorithmic model consists of a candidate generator and a verifier. In a sequential environment an execution sequence can look like $generate_1$, $verify_1$, $generate_2$, $verify_2$, where building blocks with the same index together represent one full computation cycle. It is not necessary to await the verification result of the last candidate, one can already compute the next candidate. In a concurrent environment the order of execution can for example change to $generate_1$, $generate_2$, $verify_2$, $verify_1$. There are some dependencies within each computation cycle one has to satisfy, i.e. $generate_1$ has to be executed before $verify_1$. As long as such dependencies are satisfied, building blocks can be interleaved arbitrarily. In the following a set notation is used to indicate that all building blocks within a set are executed in parallel. In a parallel environment an execution sequence can therefore look like $\{generate_1\}$, $\{generate_2, verify_1\}$, $\{verify_2\}$ or $\{generate_1\}$, $\{generate_2\}$, $\{verify_1, verify_2\}$.

Many semantics can be executed in parallel for free, if there are no dependencies between the building blocks. Now consider the preferred semantics, which has an interpretation processor as building block, namely the maximizer. Once a maximal interpretation was computed, all smaller interpretations can be excluded from the search space, since they are clearly not maximal. Hence, there is a feedback loop from a later stage of the pipeline to the candidate generator, or in other words a dependency between two building blocks. Have a look at the two execution sequences $generate_1$, $maximize_1$, $generate_2$, $maximize_2$ and $generate_1$, $generate_2$, $maximize_1$, $maximize_2$, are they equivalent? Assume the two interpretations I_1 and I_2 associated to the corresponding *generate* call by their indices and further assume $I_2 <_i maximize_1(I_1)$. Not only was a candidate generated which could have been excluded from the search space in the first sequence, but potentially a redundant interpretation is computed if $maximize_1(I_1) = maximize_2(I_2)$. Assume that a strict dependency between these building blocks is enforced, ruling out the second proposed execution sequence. Then the potential for a parallel execution for the preferred semantics was lost. Assume a weak dependency, allowing redundant interpretations from being computed, then parallelization becomes possible again, like $\{generate_1\}$, $\{generate_2, maximize_1\}$, $\{maximize_2\}$ or $\{generate_1\}$, $\{generate_2\}$, $\{maximize_1, maximize_2\}$. We have decided in favor of the second approach, since this does not interfere with the correctness of most reasoning tasks and if redundancy becomes a problem, then it is not hard to filter redundant interpretations.

Now consider the conflict-free semantics, which only consists of a generator. Further consider the execution sequence $generate_1$, $generate_2$, $generate_3$ and so on. Clearly there is a dependency between the generator to itself, since it manages the search space and excludes recently computed candidates from being generated again. Hence, for now the system is forced with the sequence $\{generate_1\}$, $\{generate_2\}$, $\{generate_3\}$ since everything else would lead to a concurrent modification of the search space, which must never happen to ensure correctness. One could now argue for a more fine-grained synchronized access to the search space, such that only one *generate* call at a time can modify it. In the big picture, this however results in sequential execution again. It is therefore necessary to find further ways of decomposition. Instead of having one generator that works with the whole search space, what if the search space is divided, allowing for the usage of multiple candidate generators, each only dealing with a restricted partition of the search space. Assume a sequence $generate_1^{\mathbf{t}(a)}$, $generate_2^{\mathbf{f}(a)}$, $generate_3^{\mathbf{u}(a)}$ for which each *generate* call deals with only a part of the search space, indicated by the superscript. Hence, $generate_1^{\mathbf{t}(a)}$ computes only interpretations I s.t. $I(a) = \mathbf{t}$, $generate_1^{\mathbf{f}(a)}$ computes only interpretations I s.t. $I(a) = \mathbf{f}$ and so on. Then the system can deal with an execution sequence like $\{generate_1^{\mathbf{t}(a)}, generate_1^{\mathbf{f}(a)}, generate_1^{\mathbf{u}(a)}\}$. If a different SAT solver state for each partition is used, then the concurrent modification problem is solved. Note that an execution sequence like $\{generate_1^{\mathbf{t}(a)}, generate_2^{\mathbf{t}(a)}\}$ would still be a problem. In the following we will discuss this approach in more detail and show that it can be generalized to allow the decomposition of the search space into arbitrarily many partitions.

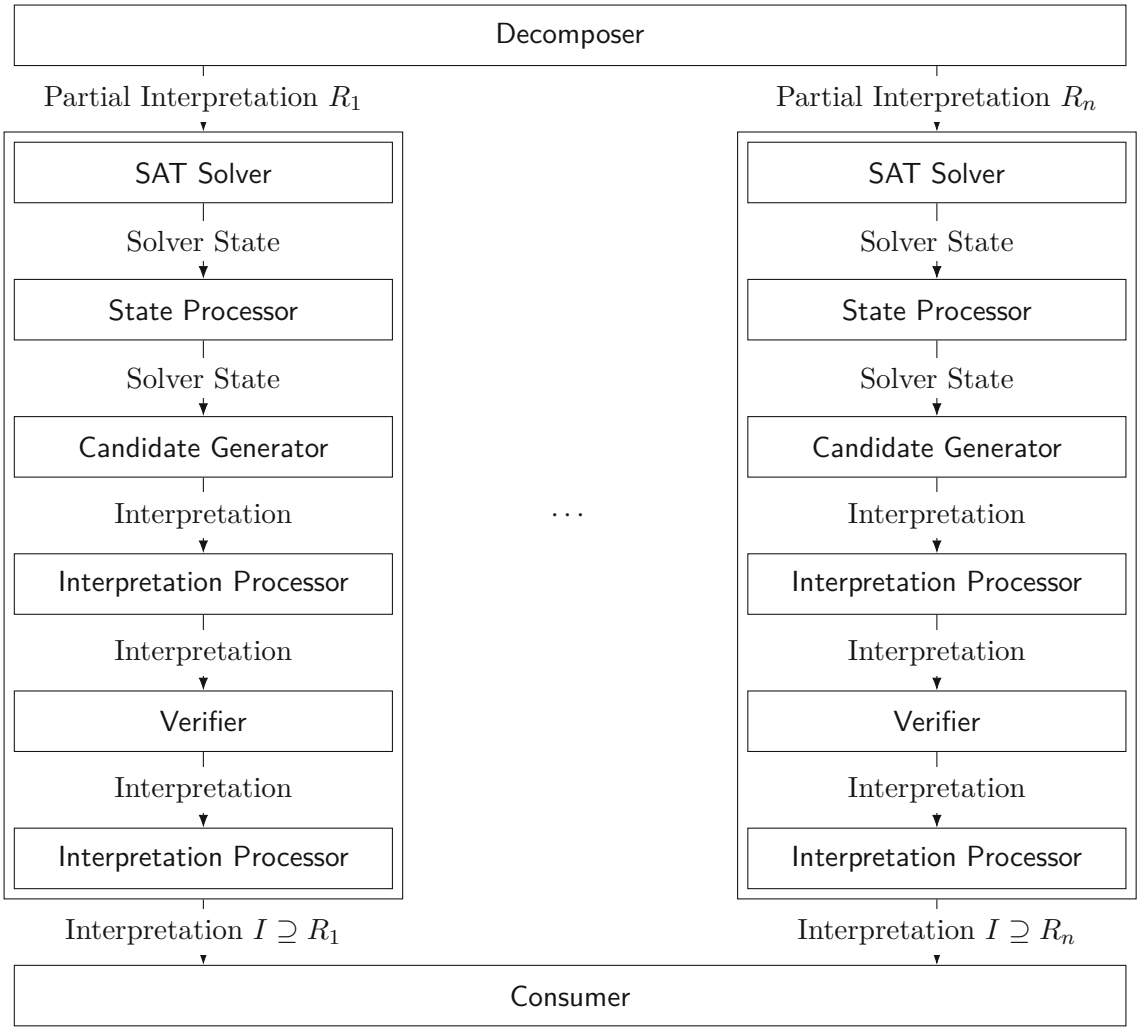


Figure 4.1: The parallel execution framework.

4.1 Framework

In principle the semantics do not change in a parallel execution environment, hence the building blocks overall stay the same. The challenge is to make sure that they can be executed concurrently with only little synchronization overhead. In addition, one further building block is introduced, namely the decomposer, which is responsible for the decomposition of the search space.

Figure 4.1 illustrates the decomposition approach. The decomposer uses some heuristics to compute a set of partial interpretations $\{R_1, \ldots, R_n\}$, these interpretations are then used to divide the search space into n parts. Each part is backed by a separate SAT solver state and restricted to only compute interpretations $I \supseteq R_i$. If the whole search space is covered by the restrictions $R_1, \ldots, R_n$ then completeness is trivial. The difficult

part is to make sure that only the restricted interpretations are computed. Since there is no general solution to this problem, it is solved on a per semantics level. The drawback of this approach is that it does not work for new semantics out of the box. Another drawback is the amount of implementation work. However, the algorithmic model of the system comes in handy again, since many building blocks are shared between semantics and thus reduce the implementation work. Also not everything has to be written from scratch, since there are synergies between the different execution models. An advantage of the per semantics level approach is the opportunity for semantics specific optimized decompositions.

4.2 Decomposer

The decomposer controls in which and in how many parts the search space is decomposed. It does so by returning a number of partial interpretations $\mathcal{R} = \{R_1, \ldots, R_n\}$, each representing a restricted part of the search space. Each restriction in $\mathcal{R}$ creates its own computation branch, which only returns interpretations I s.t. $R_i \subseteq I$ holds. Each branch has its own instance of the computation pipeline, with special restricted building blocks. The job of these building blocks is to ensure that only interpretations within the restricted search space are computed.

Definition 15 *Let $D = (A, L, C)$ be some ADF, $E \subset A$ an exclusion set and $S \subseteq A$ an argument selection, then a decomposer is a function $decompose_3 : D \times E \to 3^{S \setminus E}$ mapping ADF D to a set of partial three-valued interpretations. We further denote as $decompose_2 : D \times E \to 2^{S \setminus E}$ a decomposer which returns partial two-valued interpretations.*

If a decomposer returns all partial interpretations of some selection S, hence such that it amounts $3^{|S|}$ distinct restrictions, respectively $2^{|S|}$, completeness is trivial. The current system always computes all possible partial interpretations to ensure completeness, hence the interesting part is therefore how the set S is selected. The algorithmic framework allows for different decomposers for different semantics, hence it is possible to abuse some semantics specific information and thus skip parts of the search space entirely. The pipelines of the two-valued model and stable semantics use a two-valued decomposer for instance.

Note that often there is no need to exclude arguments from the decomposition, hence $E = \emptyset$. We then write $decompose_3(D)$ as an abbreviation for $decompose_3(D, \emptyset)$. Similar holds for $decompose_2(D)$.

Example 4 *Let $D = (\{a, b, c\}, L, C)$ be some ADF and assume that the decomposition heuristics selects $S = \{a\}$, then $decompose_3(D) = \{\{\mathbf{t}(a)\}, \{\mathbf{f}(a)\}, \{\mathbf{u}(a)\}\}$ respectively $decompose_2(D) = \{\{\mathbf{t}(a)\}, \{\mathbf{f}(a)\}\}$.*

4.2.1 Random

An easy to implement, but still useful, decomposer is the random decomposer. It randomly selects $S \subseteq A$ and thus is used to create random partitions of the search space. If an instance is run enough times using a random decomposer and by keeping track of the benchmark results, one may get some insights regarding the effect of the search space decomposition for this particular instance.

4.2.2 Most Complex Acceptance Condition

A deterministic heuristics which chooses the arguments with the most complex acceptance condition. Complex is defined with the logical complexity function lc on a purely syntactical level.

Definition 16 *Let ϕ be a propositional formula, we define the logical complexity function as follows:*

$$lc(\phi) = \begin{cases} 1 \text{ if } \phi \text{ is an atom} \\ 1 + lc(\psi) \text{ if } \phi = \neg\psi \\ 1 + lc(\psi_1) + lc(\psi_2) \text{ if } \phi = \psi_1 \circ \psi_2 \text{ with } \circ \in \{\wedge, \vee, \rightarrow, \leftrightarrow, \oplus\} \end{cases}$$

Let $D = (A, L, C)$ be an ADF, the set S of selected arguments can then be characterised by $\forall s \in S\ \forall a \in A \setminus S : lc(\varphi_a) \leq lc(\varphi_s)$.

4.2.3 Least Complex Acceptance Condition

This heuristics is the inverse of the Most Complex Acceptance Condition heuristics. Let $D = (A, L, C)$ be an ADF, the set S of selected arguments can then be characterised by $\forall s \in S\ \forall a \in A \setminus S : lc(\varphi_a) \geq lc(\varphi_s)$.

Although it does not look useful at first, since it often tends to select arguments with a trivial acceptance condition, like $\bot$ or $\top$, it is later used in the experiment chapter to illustrate the impact of the decomposition heuristics.

4.3 Candidate Generator

The principle of the candidate generator stays the same, the difference is that the generate function is restricted to only compute interpretations from a restriction of the search space. We write $generate_\sigma^R$ for the restricted version of $generate_\sigma$, where R denotes a partial interpretation. The property $R \subseteq generate_\sigma^R(\rho)$ is required to hold for arbitrary ρ until the search space is exhausted, hence until $\emptyset$ is returned.

4.3.1 Conflict-Free

The overall algorithm stays the same, what changes are the encodings added to ρ. The motivation of the modified encodings is twofold. Most important, only a part of the search space has to be computed, which makes modifications of the original encodings necessary. As a side effect, one can get rid of many unnecessary clauses. It is important to note that it is not correct to just fix the variables s^t and s^f. Consider some argument s with $\varphi_s = \bot$, by just fixing s^t for the $I(s) = \mathbf{t}$ case one would get incorrect results. Hence, it is crucial to keep the acceptance condition as well.

$$\phi^{\Gamma}_{I,s} = \begin{cases} s^t \wedge \varphi_s^{\downarrow} \text{ if } \mathbf{t}(s) \in I \\ s^f \wedge \neg\varphi_s^{\downarrow} \text{ if } \mathbf{f}(s) \in I \\ \neg s^t \wedge \neg s^f \text{ if } \mathbf{u}(s) \in I \\ (s^t \rightarrow \varphi_s^{\downarrow}) \wedge (s^f \rightarrow \neg\varphi_s^{\downarrow}) \text{ otherwise} \end{cases}$$

In principle the formula $\phi^{\Gamma}_{I,s}$ follows the same idea as the unrestricted version ϕ^{Γ}_s. It links the propositional variables representing the truth assignment of each argument to the corresponding acceptance condition. However, since a certain set of interpretations must be excluded from the computation, one can take shortcuts. Assume $I(s) = \mathbf{t}$ for instance, then it is not necessary to link s^f to $\neg\varphi_s^{\downarrow}$, since interpretations where $I(s) = \mathbf{f}$ holds are not considered anyway. In practice, this also means that if $I(s) = \mathbf{t}$ or $I(s) = \mathbf{f}$ is fixed, the optimized version of the Tseitin transformation can be used for $\varphi_s^{\downarrow}$, since the acceptance condition then only appears in either positive or negative polarity in $\phi^{\Gamma}_{I,s}$. Hence, we can skip additional clauses, besides the apparent ones.

$$\phi^{pr}_{I,s} = \begin{cases} \bigwedge\limits_{(s,r)\in L} p^s_r \text{ if } \mathbf{t}(s) \in I \\ \bigwedge\limits_{(s,r)\in L} \neg p^s_r \text{ if } \mathbf{f}(s) \in I \\ \top \text{ if } \mathbf{u}(s) \in I \\ (s^t \rightarrow \bigwedge\limits_{(s,r)\in L} p^s_r) \wedge (s^f \rightarrow \bigwedge\limits_{(s,r)\in L} \neg p^s_r) \text{ otherwise} \end{cases}$$

Again, the formula $\phi^{pr}_{I,s}$ follows the same idea as its unrestricted version ϕ^{pr}_s. For each argument s the formula ϕ^{pr}_s forces the truth assignments of the link variables, used in the acceptance conditions of the children of s, to match the truth assignment of s. Since this is only relevant if s is not undecided, it immediately follows that one can get rid of this encoding if $I(s) = \mathbf{u}$ holds. For the decided cases the link variables can be fixed directly, according to the truth value of s in I.

Proposition 12 *Let $D = (A, L, C)$ be some ADF, R some partial interpretation over D and further let $\phi^{\Gamma}_{R,s} \in \rho$ and $\phi^{pr}_{R,s} \in \rho$ hold for each $s \in A$. Then $\text{generate}^R_{cf}(\rho) = I \neq \emptyset$ implies $R \subseteq I$ and I being conflict-free.*

Proof. Assume $R \not\subseteq I$, i.e. one of the following cases holds:

Algorithm 4.1: $generate^{R}_{cf}(\rho)$

```
(τ, sat) ← SAT(ρ)
if sat then
    I ← Extract(τ)
    ρ ← ρ ∧ φ^I_≠
    return I
end
return ∅
```

- $\mathbf{u}(s) \in R$ but $\mathbf{u}(s) \notin I$: Observe $\neg s^t \wedge \neg s^f$ in ρ, which is only satisfied if $\mathbf{u}(s) \in I$.
- $\mathbf{t}(s) \in R$ but $\mathbf{t}(s) \notin I$: Observe $s^t \wedge \varphi_s^{\downarrow}$ in ρ, which is only satisfied if $\mathbf{t}(s) \in I$.
- $\mathbf{f}(s) \in R$ but $\mathbf{t}(s) \notin I$: Observe $s^f \wedge \neg\varphi_s^{\downarrow}$ in ρ, which is only satisfied if $\mathbf{f}(s) \in I$.

Hence, since all cases are contradicting it follows $R \subseteq I$.

It remains to show that I is conflict-free. By looking at the definition of conflict-free there are two cases to consider:

- $\mathbf{t}(s) \in I$: If $\mathbf{t}(s) \in R$ holds then from $s^t \wedge \varphi_s^{\downarrow}$ it follows that $\varphi_s^{\downarrow}$ is satisfiable. Since there is the formula $\phi^{pr}_{I,s}$, which assigns the link variables according to I, it follows that $\varphi_s^{\downarrow}[I]$ is also satisfiable.
- $\mathbf{f}(s) \in I$: If $\mathbf{f}(s) \in R$ holds then from $s^f \wedge \neg\varphi_s^{\downarrow}$ it follows that $\varphi_s^{\downarrow}$ is refutable. Since there is the formula $\phi^{pr}_{I,s}$, which assigns the link variables according to I, it follows that $\varphi_s^{\downarrow}[I]$ is also refutable.

It is sufficient to only consider the cases where $I(s) = R(s)$, since for all other cases the encodings are equal to its unrestricted versions. Hence, since both cases hold I is conflict-free. □

4.3.2 Two-Valued Model

The overall algorithm stays the same, what changes are the encodings added to ρ. Since only two-valued interpretations are computed, it does not make sense to consider three-valued partitions of the search space. Hence, the restrictions are expected to be two-valued.

$$\phi^{mod}_{I,s} = \begin{cases} s^t \wedge \varphi_s \text{ if } \mathbf{t}(s) \in I \\ s^f \wedge \neg\varphi_s \text{ if } \mathbf{f}(s) \in I \\ (s^t \leftrightarrow \varphi_s) \wedge (\neg s^t \rightarrow s^f) \text{ otherwise} \end{cases}$$

The formula $\phi^{mod}_{I,s}$ has similarities with $\phi^{\Gamma}_{I,s}$, the main difference is that the undecided case can be skipped. In order to restrict the search space, it is not sufficient to only fix s^t or s^f, it is necessary to also keep the acceptance conditions. Otherwise cases like $I(s) = \mathbf{t}$ with $\varphi^{\downarrow}_s = \bot$ would return interpretations, although in such a case no interpretation must be returned at all.

Algorithm 4.2: $generate^R_{mod}(\rho)$

```
(τ, sat) ← SAT(ρ)
if sat then
    I ← Extract(τ)
    ρ ← ρ ∧ φ^I_≠
    return I
end
return ∅
```

Proposition 13 *Let $D = (A, L, C)$ be some ADF, R some two-valued partial interpretation over D and further let $\phi^{mod}_{R,s} \in \rho$ hold for each $s \in A$. Then $generate^R_{mod}(\rho) = I \neq \emptyset$ implies $R \subseteq I$ and I being a two-valued model.*

Proof. Assume $R \not\subseteq I$, i.e. one of the following cases holds:

- $\mathbf{t}(s) \in R$ but $\mathbf{t}(s) \notin I$: Observe $s^t \wedge \varphi^{\downarrow}_s$ in ρ, which is only satisfied if $\mathbf{t}(s) \in I$.
- $\mathbf{f}(s) \in R$ but $\mathbf{t}(s) \notin I$: Observe $s^f \wedge \neg\varphi^{\downarrow}_s$ in ρ, which is only satisfied if $\mathbf{f}(s) \in I$.

Hence, since all cases are contradicting it follows $R \subseteq I$.

It remains to show that I is a two-valued model. We show $I(a) = \mathbf{t}$ iff $\varphi_s[I]$ is satisfiable as $I(a) = \mathbf{t}$ implies $\varphi_s[I]$ is satisfiable and $I(a) = \mathbf{f}$ implies $\varphi_s[I]$ is not satisfiable.

- $\mathbf{t}(s) \in I$: If $\mathbf{t}(s) \in R$ holds then from $s^t \wedge \varphi_s$ it follows that φ_s is satisfiable. Since we only deal with two-valued models and therefore with φ_s directly without link variables, it follows that $\varphi_s[I]$ is satisfiable.
- $\mathbf{f}(s) \in I$: If $\mathbf{f}(s) \in R$ holds then from $s^f \wedge \neg\varphi_s$ it follows that $\neg\varphi_s$ is satisfiable. Since we only deal with two-valued models and therefore with φ_s directly without link variables, it follows that $\neg\varphi_s[I]$ is satisfiable, hence $\varphi_s[I]$ is not satisfiable.

It is sufficient to only consider the cases where $I(s) = R(s)$, since for all other cases the encodings are equal to its unrestricted versions. Hence, I is a two-valued model. □

4.3.3 Grounded

The grounded generator works differently than the other generators. It is a direct implementation of the fixpoint operator and thus has its semantical properties not encoded in the search space ρ. The main problem for parallelization is that there is only one unique grounded interpretation. So, if the search space is decomposed into n distinct partitions, then $n-1$ of them do not return an interpretation. Although the grounded semantics takes no advantage from the parallelization framework, the notion of restricted semantics still has its application for many reasoning tasks, as we will show later. Hence, in order to restrict the grounded generator, shortcuts are introduced by checking the recent assignments. Assume that $I(s) = \mathbf{t}$ is assigned, then if neither $\mathbf{f}(s) \in R$ nor $\mathbf{u}(s) \in R$ holds continue, otherwise stop and return $\emptyset$.

Algorithm 4.3: $generate^{R}_{grd}(\rho)$

```
1  I_new ← I_u
2  I_old ← I_u
3  repeat
4  |  I_old ← I_new
5  |  ρ ← ρ ∪ {s^t | I_old(s) = t} ∪ {s^f | I_old(s) = f}
6  |  for s ∈ A with I_old(s) = u do
7  |  |  if ¬SAT(ρ ∪ {¬φ_s^↓}) then
8  |  |  |  if f(s) ∈ R or u(s) ∈ R then
9  |  |  |  |  ρ ← ρ ∪ {⊥}
10 |  |  |  |  return ∅
11 |  |  |  end
12 |  |  |  I_new(s) ↦ t
13 |  |  else if ¬SAT(ρ ∪ {φ_s^↓}) then
14 |  |  |  if t(s) ∈ R or u(s) ∈ R then
15 |  |  |  |  ρ ← ρ ∪ {⊥}
16 |  |  |  |  return ∅
17 |  |  |  end
18 |  |  |  I_new(s) ↦ f
19 |  |  else
20 |  |  |  I_new(s) ↦ u
21 |  |  end
22 |  end
23 until I_new = I_old
24 ρ ← ρ ∪ {⊥}
25 if ∃v(s) ∈ R : v(s) ∉ I_new with v ∈ {t, f, u} then
26 |  return ∅
27 end
28 return I_new
```

Note that shortcuts are only introduced for the decided cases, since these assignments are fixed and cannot change in the following iterations. It is also necessary to check if the computed interpretation is consistent with the restriction R before it is returned. This rules out cases where some argument is decided in the restriction R but undecided in I_{new}.

Proposition 14 *Let $D = (A, L, C)$ be some ADF, R some partial interpretation over D and $\rho = \bigwedge_{s \in A}(\phi_s^\Gamma \wedge \phi_s^{pr})$. Then $generate_{grd}^R(\rho) = I \neq \emptyset$ implies $R \subseteq I$ and I being grounded.*

Proof. Assume $R \not\subseteq I$, then at some point $\emptyset$ is returned because of the introduced shortcuts. This is however a contradiction to $I \neq \emptyset$, hence $R \subseteq I$ holds. Since the rest of the algorithm is equal to the unrestricted version, I is grounded. □

4.4 Verifier

A verifier does not compute any interpretation, it just checks some property and acts as a filter. Thus, there is no need for any special handling for the restricted case, since the verification process remains the same. However, two additional verifiers are needed for the concurrent framework. Assume some maximizer that only computes interpretations under some restriction R, hence each maximal interpretation I has to satisfy $R \subseteq I$. Now assume some ADF $D = (\{a\}, \emptyset, \{\varphi_a = \top\})$ and restriction $R = \{\mathbf{u}(a)\}$, then some maximizer restricted to R can only return the interpretation $I = \{\mathbf{u}(a)\}$. It is not hard to see that I is however not maximal in D, since clearly argument a can be decided. Hence, in order to rule out interpretations which are only maximal in the restricted case, but not in the unrestricted, additional verifiers are needed for the naive and preferred semantics.

4.4.1 Naive

Naive interpretations are maximally conflict-free. The following algorithm assumes that the interpretations to verify are already conflict-free, so it remains to check if they are maximal. This can be done by trying to decide one further undecided argument. If it is possible to do so, the interpretation was clearly not maximal and therefore not naive.

Algorithm 4.4: $verify_{nai}(I)$

1 $A \leftarrow \{s^t \mid I(s) = \mathbf{t}\} \cup \{s^f \mid I(s) = \mathbf{f}\}$
2 $\phi_> \leftarrow toggle \vee \bigvee_{I(s)=\mathbf{u}} s^t \vee s^f$
3 $\nu \leftarrow \nu \cup \{\phi_>\}$
4 **return** $\neg SAT(\nu, A \cup \{\neg toggle\})$

Algorithm 4.4 employs the techniques of assumption based SAT calls and interpretation independent encodings. The idea is to reduce the overhead of each $\mathit{verify}(I)$ call by reusing the same SAT solver state. This is especially useful since the formulas ϕ_s^Γ and ϕ_s^{pr} must be added to the verification state ν. So by reusing the solver state, these encodings have to only be computed once. The problematic part is the search for a larger interpretation, since this is inherently interpretation specific. Since an interpretation specific encoding makes the solver state not reusable, we use the idea of a fresh propositional variable that acts as a toggle. Let formula $\phi_>$ be defined in a way such that it is always satisfiable via the toggle variable. It is then possible to add clauses to the verification state which are only activated once, by explicitly assuming the toggle variable as false. Hence, these clauses do not interfere with subsequent $\mathit{verify}(I)$ calls.

Proposition 15 *Let $D = (A, L, C)$ be some ADF, $\nu \supseteq \bigcup_{s \in A}\{\phi_s^\Gamma, \phi_s^{pr}\}$ the verification state and I be some conflict-free interpretation, then $\mathit{verify}_{nai}(I) = \mathit{true}$ iff I is naive in D.*

Proof. Assume $\mathit{verify}_{cf}(I) = \mathit{true}$ holds. Then $SAT(\nu, A \cup \{\neg \mathit{toggle}\})$ must be false. Since *toggle* was assumed to be false, the clause $\phi_>$ could not be satisfied by deciding another argument. Hence, since no further arguments could be decided, the interpretation was maximal and by assumption conflict-free, therefore naive.

Assume I is naive in D. The *toggle* variable again is assumed to be false, hence $\phi_>$ is only satisfiable if another argument can be decided. This would contradict the assumption that I is naive, hence $SAT(\nu, A \cup \{\neg \mathit{toggle}\})$ returns false from which $\mathit{verify}_{cf}(I) = \mathit{true}$ follows. □

4.4.2 Preferred

An interpretation is preferred if it is maximally admissible. The algorithm expects that the given interpretation is already admissible, so it suffices to check if it is maximal. The same techniques are used for the preferred verifier as we did for the naive verifier. The main difference is to make sure that only admissible interpretations are considered as larger in respect to the information ordering $<_i$. This can be done by also adding the formulas ϕ_s^{bip} and $\phi_s^{\Gamma_?}$ to the verification state, besides ϕ_s^Γ and ϕ_s^{pr}.

Algorithm 4.5: $\mathit{verify}_{prf}(I)$

1 $A \leftarrow \{s^t \mid I(s) = \mathbf{t}\} \cup \{s^f \mid I(s) = \mathbf{f}\}$
2 $\phi_> \leftarrow \mathit{toggle} \vee \bigvee_{I(s)=\mathbf{u}} s^t \vee s^f$
3 $\nu \leftarrow \nu \wedge \phi_>$
4 **return** $\neg SAT(\nu, A \cup \{\neg \mathit{toggle}\})$

Proposition 16 *Let $D = (A, L, C)$ be some ADF, $\nu \supseteq \bigcup_{s \in A} \{\phi_s^{\Gamma}, \phi_s^{pr}, \phi_s^{bip}, \phi_s^{\Gamma_?}\}$ the verification state and I be some admissible interpretation, then $verify_{prf}(I) = true$ iff I is preferred in D.*

Proof. Analogous to the naive verifier. □

4.4.3 Stable

One application of the restricted grounded generator is the stable verifier. It is possible to take advantage of the shortcuts introduced in Algorithm 4.3 by slightly modifying the stable verifier. The overall algorithm is basically the same as the one introduced in the previous chapter. Now the check if I_{grd} of D^I is I is computed indirectly. If I is not stable, then the shortcuts make sure that the grounded interpretation restricted to I is $\emptyset$, since it has to differ by definition.

Algorithm 4.6: $verify_{stb}(I)$

```
1 I_grd ← generate^I_grd(D^I)
2 return I_grd ≠ ∅
```

Proposition 17 *Let D be some ADF and I be some two-valued interpretation, then $verify_{stb}(I) = true$ iff I is stable in D.*

Proof. Assume that I is stable, then by definition the grounded interpretation of I^D is I, thus $I_{grd} = I$ and clearly $I_{grd} \neq \emptyset$.

Assume that I is not stable, then by definition the grounded interpretation of I^D is not I, thus $generate^I_{grd}(D^I) = \emptyset$ and clearly $I_{grd} = \emptyset$. □

4.5 Interpretation Processor

Although not generating interpretations, since an interpretation processor takes, modifies and returns interpretations, it is necessary to ensure that each returned interpretation is satisfying the restriction R. It is already known from the candidate generation how to restrict the search space to only compute interpretations I with $R \subseteq I$. Hence, the same techniques are used as for the candidate generator. Note that it was already discussed in the verifier section that this only returns maximal interpretations relative to some restriction R.

4.5.1 Conflict-Free Maximizer

By comparing Algorithm 4.7 with Algorithm 3.7, one can observe that indeed the only difference is the initialization of φ. This is not surprising, since conceptually there is no

real difference between the algorithms. The only additional thing to consider is to make sure that each returned interpretation I satisfies $R \subseteq I$. In order to do so, only φ has to be modified.

Algorithm 4.7: $maximize^R_{cf}(I)$

$\varphi \leftarrow \phi^I_> \wedge \bigwedge_{s \in A}(\phi^\Gamma_{R,s} \wedge \phi^{pr}_{R,s})$
$I_{max} \leftarrow I$
$(w, sat) \leftarrow SAT(\varphi)$
while *sat* **do**
  $I_{max} \leftarrow Extract(w)$
  $\varphi \leftarrow \varphi \wedge \phi^I_>$
  $(w, sat) \leftarrow SAT(\varphi)$
end
return $(I_{max}, \phi^{I_{max}}_{\not<})$

Proposition 18 *Let D be some ADF and I be some conflict-free interpretation, then interpretation J with $maximize_{cf}(I) = (J, \phi^J_{\not<})$ is maximally conflict-free in D relative to R and $R \subseteq J$.*

Proof. Observe that $R \subseteq I_{max}$ holds for each I_{max} by the definition of φ. Hence, $R \subseteq J$ must hold. Further observe that, again by the definition of φ, each I_{max} is conflict-free, hence J must be conflict-free. Since the maximization process works analogous to Algorithm 3.7 the interpretation J must be maximal relative to R. □

4.5.2 Admissible Maximizer

Again, compare Algorithm 4.8 with Algorithm 3.8 and observe that just the definition of φ has to change in order to only compute interpretations I with the restriction $R \subseteq I$.

Algorithm 4.8: $maximize^R_{adm}(I)$

$\varphi \leftarrow \phi^I_> \wedge \bigwedge_{s \in A}(\phi^\Gamma_{R,s} \wedge \phi^{pr}_{R,s} \wedge \phi^{\Gamma_?}_s \wedge \phi^{bip}_s)$
$I_{max} \leftarrow I$
$(w, sat) \leftarrow SAT(\varphi)$
while *sat* **do**
  $I_{max} \leftarrow Extract(w)$
  $\varphi \leftarrow \varphi \wedge \phi^I_>$
  $(w, sat) \leftarrow SAT(\varphi)$
end
return $(I_{max}, \phi^{I_{max}}_{\not<})$

Proposition 19 *Let D be some ADF and I be some admissible interpretation, then interpretation J with $maximize_{cf}(I) = (J, \phi^J_{\not\subset})$ is maximal admissible in D relative to R and $R \subseteq J$.*

Proof. Analogous to the conflict-free maximizer. □

4.6 Putting Things Together

In this section the previously defined building blocks are wired together in a way that allows concurrent execution.

Definition 17 *Let $\mathcal{T} = \{T_1, \ldots, T_n\}$ denote a thread-pool of size n and $\mathcal{Q}$ denote a task queue. Assume the existence of some mechanism removing a thread $T \in \mathcal{T}$, some task $f \in \mathcal{Q}$ and then executes f on T. On completion of f on T, T is added back to $\mathcal{T}$. If either $\mathcal{T} = \emptyset$ or $\mathcal{Q} = \emptyset$ the mechanism waits. We define some statement 'run f' which takes some task f and adds it to $\mathcal{Q}$.*

Definition 17 models the concurrent execution behaviour and introduces the *run* statement. The only thing this statement does is adding the task on its right-hand side to the execution queue. It is then up to some background mechanism to determine when and how this task is executed. The important takeaway is that *run* terminates immediately, hence it does not wait until the task on the right-hand side is completed. The workings of the execution mechanism are intentionally kept vague. There are various scheduling and queueing strategies one can use, we handle them as an implementation detail and therefore use the *run* statement as an abstraction.

In contrast to the sequential framework, the various execution stages are now put into separate algorithms. This is important since we use them as targets for the *run* statement. Note that the superscript of each step indicates a hardcoded reference to some structure. Take the set $\mathcal{I}$ for instance, which is referenced by the last step of the pipeline to have some consumer for the computed interpretations. Note that while the set $\mathcal{I}$ is global, the set $\mathcal{U}$ of update formulas is only shared by building blocks within the same restricted branch.

Algorithm 4.9: $enumerate_\sigma(D)$

```
1 I ← ∅
2 R ← decompose^3(D)
3 foreach restriction R ∈ R do
4 |   run initializationStep^R_σ(D)
5 end
6 return I
```

Algorithm 4.9 is the entry point of the concurrent execution. As a first step, the decomposer is applied to the given ADF D, which results in a set of partial interpretations $\mathcal{R}$. Then each $R \in \mathcal{R}$ is used to spawn its own restricted computation branch and thus to enumerate each restricted search space concurrently.

Algorithm 4.10: $initializationStep_{\sigma}^{R}(D)$

```
1 ρ ← processState(∅)
2 U ← ∅
3 run generationStep_σ^{R,U}(ρ)
```

Since each restricted search space has its own SAT solver state, the state processor has to be called for each of them. However, since Algorithm 4.10 is called via *run*, multiple states can be processed in parallel. The candidate computation can begin once the initial state processing is done. The *run* statement is used to execute the next stage of the pipeline, this way the current thread is returned to the thread pool and the scheduling mechanism can decide which task to handle next.

Algorithm 4.11: $generationStep_{\sigma}^{R,\mathcal{U}}(\rho)$

```
1 ρ ← ρ ∪ U
2 U ← ∅
3 I ← generate^R(ρ)
4 if I ≠ ∅ then
5 |   run unverifiedProcessingStep_σ^{R,U}(I)
6 |   run generationStep_σ^{R,U}(ρ)
7 end
```

Algorithm 4.11 has access to the shared data structure $\mathcal{U}$. This structure is also referenced by interpretation processors and is used to communicate search space updates to the candidate generator. Since multiple interpretation processors may add propositional formulas to $\mathcal{U}$ in parallel, the structure $\mathcal{U}$ is assumed to be synchronized. A synchronized data structure ensures a consistent behaviour within a multi-threaded environment, this is usually achieved via locks. Observe that there is only one instance of Algorithm 4.11 running at the same time, because at this stage of the pipeline it is never executed by any other algorithm besides itself. Hence, it is safe to add the formulas in $\mathcal{U}$ to ρ at the beginning of the algorithm, without corrupting the search space by concurrent modifications.

This algorithm is also the reason redundant interpretations may be computed. Algorithm 4.11 does not wait until some interpretation processor at a later stage completes, it just runs *unverifiedProcessingStep* with the computed candidate and then itself again. If *generationStep* is then executed again, it just adds whatever is in $\mathcal{U}$ to the search space. It does not matter if the interpretation processor is already done or not. Note that this

may also happen in a single-threaded environment, since *run* gives no guarantees when a task is executed, it is perfectly fine for tasks to be executed in an order that does not match the order of *run* statements.

More formally, a candidate generator can compute a sequence of interpretations I_1, I_2, ..., I_k before the effects of I_1 to the search space are propagated back via $\mathcal{U}$. This can then lead to the computation of redundant interpretations if the *process* function is not one-to-one, i.e. $\exists I, J$ with $I \neq J$ such that $process(I) = process(J)$. In order to solve this problem it would require to wait for updates of the search space before the computation of the next candidate begins. This would have a negative impact on the throughput of the pipeline. More importantly, redundancy does not interfere with the reasoning tasks we are interested in. Hence, we have decided to give up uniqueness for the sake of a higher throughput.

Algorithm 4.12: $unverifiedProcessingStep_{\sigma}^{R,\mathcal{U}}(I)$

```
1 (J, ψ) ← processUnverified(I)
2 U ← U ∪ {ψ}
3 run verificationStep_σ^R(J)
```

The first of the two interpretation processors is run before the verifier and is therefore denoted as *unverifiedProcessingStep*, since it deals with unverified candidates. It has access to the already mentioned shared data structure $\mathcal{U}$, which it uses to communicate search space updates back to the previous stage.

Algorithm 4.13: $verificationStep_{\sigma}^{R}(I)$

```
1 if verify(I) then
2 |   run verifiedProcessingStep_σ^{R,U,I}(I)
3 end
```

Algorithm 4.13 acts as a filter, the next stage of the pipeline is only executed if the verification is successful.

Algorithm 4.14: $verifiedProcessingStep_{\sigma}^{R,\mathcal{U},\mathcal{I}}(I)$

```
1 (J, ψ) ← processVerified(I)
2 U ← U ∪ {ψ}
3 I ← I ∪ {J}
```

The processing of verified interpretations is the last step of the pipeline. Algorithm 4.14 has a reference to the global set of computed interpretations $\mathcal{I}$. The framework does not synchronize the access to $\mathcal{I}$, hence thread-safety is established by the structure $\mathcal{I}$ itself. At this stage of the pipeline, J represents an interpretation of semantics σ, the computation is therefore completed once J is added to $\mathcal{I}$.

In the following, it is illustrated how the building blocks interact together for each semantics. Again, the following sections do not show distinct algorithms, but rather how the previously defined framework is configured for each semantics. In contrast to Algorithm 3.9, the parallel framework does not work with defaults for missing building blocks, but skips them entirely. Hence, the wiring of the steps may differ, depending on which steps are necessary.

4.6.1 Conflict-Free

Although it may look different because of the different execution model, from a conceptual standpoint the conflict-free algorithm is the same as in the previous chapter. There is no need to change any building blocks.

Algorithm 4.15: $\mathit{initializationStep}^{R}_{cf}(D)$

```
1 ρ ← ⋃_{s∈A} {φ^Γ_{R,s}, φ^{pr}_{R,s}}
2 run generationStep^{R,I}_{cf}(ρ)
```

Algorithm 4.15 represents the initialization step. No state processor is used, it therefore only encodes the basic properties of conflict-free interpretations.

Algorithm 4.16: $\mathit{generationStep}^{R,\mathcal{I}}_{cf}(\rho)$

```
1 I ← generate^R_{cf}(ρ)
2 if I ≠ ∅ then
3 |   I ← I ∪ {I}
4 |   run generationStep^{R,I}_{cf}(ρ)
5 end
```

Algorithm 4.16 then exhaustively computes conflict-free interpretations until the search space is exhausted. Since there is no further step for the conflict-free semantics, the computed interpretations are directly added to the shared set of interpretations $\mathcal{I}$. Notice an important subtlety here. In practice the shared set $\mathcal{I}$ is usually bounded, to prevent the computation of all interpretations if no one is consuming them. Hence, at one point $\mathcal{I}$ is full and every attempt to add another interpretation blocks the current thread until some consumer starts draining $\mathcal{I}$. Now imagine what would happen if the run statement is used at line $\mathcal{I} \leftarrow \mathcal{I} \cup \{I\}$. Then the addition of I to $\mathcal{I}$ would be delayed until the scheduling mechanism decides to run it. In some worst case scenario this could lead to the pollution of the task queue, or unnecessarily delays computed interpretations for the consumer. Therefore, interpretations are always added to $\mathcal{I}$ in the same thread as the last step of the computation pipeline.

4.6.2 Naive

The concurrent execution differs from the sequential one because of the additional verification step.

Algorithm 4.17: $initializationStep^{R}_{nai}(D)$

```
1 ρ ← ⋃_{s∈A} {φ^Γ_{R,s}, φ^pr_{R,s}}
2 run generationStep^{R,U}_{nai}(ρ)
```

The first step of the computation is the same as for the conflict-free semantics, as Algorithm 4.17 shows.

Algorithm 4.18: $generationStep^{R,\mathcal{U}}_{nai}(\rho)$

```
1 U' ← U
2 ρ ← ρ ∪ U'
3 U ← U \ U'
4 I ← generate^R_{cf}(ρ)
5 if I ≠ ∅ then
6 |   run unverifiedProcessingStep^{R,U}_{nai}(I)
7 |   run generationStep^{R,U}_{nai}(ρ)
8 end
```

Algorithm 4.18 already differs from the conflict-free case. Since there is an interpretation processor for the maximization of conflict-free interpretations, the update formulas $\mathcal{U}$ must be handled. An intermediate set $\mathcal{U}'$ is necessary to neither lose any updates nor add them twice. If the algorithm would add $\mathcal{U}$ directly to ρ and then set it to $\emptyset$ afterwards, all updates which were added between those two instructions are lost. If it works like above, but still adds $\mathcal{U}$ directly to ρ, then the updates which are added after the initialization of $\mathcal{U}'$ are added twice. The wiring also differs, since there is a follow-up step after the candidate generation. The generated interpretations are not immediately returned, but are handed to the next computation step.

Algorithm 4.19: $unverifiedProcessingStep^{R,\mathcal{U}}_{nai}(I)$

```
1 (J, ψ) ← maximize^R_{cf}(I)
2 U ← U ∪ {ψ}
3 run verificationStep^{R,I}_{nai}(J)
```

Algorithm 4.19 takes a conflict-free interpretation I and calls the restricted conflict-free maximizer on it. The resulting update formula is then communicated back to the previous step via the shared data structure $\mathcal{U}$. Note that the interpretation J is only maximal relative to restriction R. Hence, it remains to verify if it is also maximal in ADF D.

Algorithm 4.20: $verificationStep^{R,\mathcal{I}}_{nai}(I)$

```
1 if verify_nai(I) then
2 |  I ← I ∪ {I}
3 end
```

Algorithm 4.20 is the first step that differs from the sequential algorithm for naive interpretations. If not dealing with restricted search spaces, there is no need for an extra verification step, since the maximizer computes naive interpretations directly. In the restricted case we however have to filter non-maximal results first.

Example 5 *Consider the ADF* $D = (\{a\}, L, \{\varphi_a = \top\})$. *Let* $\mathcal{T} = \{T_1, T_2\}$ *denote the thread pool and* $\mathcal{Q}$ *the task queue, further assume a mechanism which executes tasks in a first-in-first-out manner. In the following table for each timestamp at least one task finishes. We further abbreviate the names of the algorithms to cope with the limited space.*

Time	$\mathbf{T_1}$	$\mathbf{T_2}$	$\mathcal{Q}$
1	$initializationStep^{\mathbf{t}}$	$initializationStep^{\mathbf{f}}$	$\{initializationStep^{\mathbf{u}}\}$
2	$initializationStep^{\mathbf{u}}$	$initializationStep^{\mathbf{f}}$	$\{generationStep^{\mathbf{t}}\}$
3	$initializationStep^{\mathbf{u}}$	$generationStep^{\mathbf{t}}$	$\{generationStep^{\mathbf{f}}\}$
4	$generationStep^{\mathbf{f}}$	$generationStep^{\mathbf{t}}$	$\{generationStep^{\mathbf{u}}\}$
5	$generationStep^{\mathbf{u}}$	$generationStep^{\mathbf{t}}$	$\emptyset$
6	$unverifiedProcessingStep^{\mathbf{u}}$	$generationStep^{\mathbf{t}}$	$\{generationStep^{\mathbf{u}}\}$
7	$generationStep^{\mathbf{u}}$	$generationStep^{\mathbf{t}}$	$\{verificationStep^{\mathbf{u}}\}$
8	$verificationStep^{\mathbf{u}}$	$generationStep^{\mathbf{t}}$	$\emptyset$
9	$verificationStep^{\mathbf{u}}$	$unverifiedProcessingStep^{\mathbf{t}}$	$\{generationStep^{\mathbf{t}}\}$
10	$generationStep^{\mathbf{t}}$	$unverifiedProcessingStep^{\mathbf{t}}$	$\emptyset$
11	-	$verificationStep^{\mathbf{t}}$	$\emptyset$

An almost trivial example was chosen to illustrate the parallel execution of the naive semantics. This has several reasons, one of them is space, since the runtime profile explodes even for the smallest ADFs. However, it also illustrates that it is hard to reason about the runtime behaviour of parallel algorithms. Above table is just one possible execution order, hand-crafted to fit on the page, but there are many more even for such a simple example. Hence, reasoning about properties like termination or correctness becomes way harder compared to the sequential execution model.

Since the scheduling mechanism is deterministic, it is easier to follow the steps of table above. The execution begins with the decomposition, since there is only one argument it holds $\mathcal{R} = \{\mathbf{t}(a), \mathbf{f}(a), \mathbf{u}(a)\}$. *Then the initialization steps are executed, since there are three of them, but just two threads, one is delayed and put into the task queue* $\mathcal{Q}$. *Thread* T_1 *is the first to finish, immediately assigning it to the other initialization task. At timestamp 4 thread* T_1 *finishes the partition* $\mathbf{f}(a)$, *since there is no such candidate,*

hence no new tasks are scheduled for execution. At timestamp 5 thread T_1 generates the candidate $I = \mathbf{u}(a)$ and schedules the maximization task, which is executed immediately. The maximizer returns I again, which is maximal relative to the restriction $\{\mathbf{u}(a)\}$. There are no further candidates within the partition $\mathbf{u}(a)$, hence the generation task on thread T_1 terminates without spawning new tasks. The candidate generation on thread T_2 returns $I' = \{\mathbf{t}(a)\}$ at timestamp 9 and immediately executes the maximization thread, since there are no prior tasks in $\mathcal{Q}$. The verification step on thread T_1 discards interpretation I, since it is clearly not maximal. The upcoming generation thread then also terminates since the search space is exhausted. On thread T_2 the interpretation I' was handed to the verification step. Since I' is naive, and therefore not discarded, it is added to the set $\mathcal{I}$.

4.6.3 Two-Valued Model

The algorithmic model for the two-valued semantics is similar to the conflict-free semantics. This is not surprising and also analogous to the sequential algorithmic model.

Algorithm 4.21: $initializationStep^{R}_{mod}(D)$

```
1 ρ ← ⋃_{s∈A} {φ^{mod}_{R,s}}
2 run generationStep^{R,I}_{mod}(ρ)
```

Algorithm 4.21 is the starting point of the computation. There currently is no need for a state processor, so only the semantics specific properties are encoded into search space ρ.

Algorithm 4.22: $generationStep^{R,\mathcal{I}}_{mod}(\rho)$

```
1 I ← generate^{R}_{mod}(ρ)
2 if I ≠ ∅ then
3 |   I ← I ∪ {I}
4 |   run generationStep^{R,I}_{mod}(ρ)
5 end
```

Once the search space is initialized, the computation of two-valued models is straight forward, as there is no need for further building blocks, illustrated by Algorithm 4.22. Again, notice that since there are no further steps involved in the computation, the two-valued models I are added to $\mathcal{I}$ in the same thread as they were generated. This prevents some undesired side-effects, as discussed in more detail in the conflict-free section.

4.6.4 Admissible

The following algorithm makes use of the k-bipolar state processor, the admissibility property is therefore directly encoded into ρ. Hence, there is no need for an additional admissibility verification step, leaving us with only two steps in the computation pipeline.

Algorithm 4.23: $initializationStep^{R}_{adm}(D)$

```
1 ρ ← ⋃_{s∈A} {φ^Γ_{R,s}, φ^{pr}_{R,s}}
2 ρ ← process_{kbip}(ρ)
3 run generationStep^{R,I}_{adm}(ρ)
```

Algorithm 4.23 initializes the search space with the conflict-free encodings. Then the k-bipolar optimization is encoded into ρ by the respective state processor. This results in a search space that only returns admissible interpretations.

Algorithm 4.24: $generationStep^{R,\mathcal{I}}_{adm}(\rho)$

```
1 I ← generate^R_{cf}(ρ)
2 if I ≠ ∅ then
3 |   I ← I ∪ {I}
4 |   run generationStep^{R,I}_{adm}(ρ)
5 end
```

Algorithm 4.24 performs the exhaustive generation of admissible interpretations, analogous to the conflict-free or two-valued model semantics. As always, since there are no further steps in the pipeline, the interpretations are added to $\mathcal{I}$ in the current thread.

4.6.5 Preferred

Conceptually the preferred semantics is similar to the naive semantics, hence similar problems occur. In contrast to the sequential algorithm, there is now need for a verification step. The problem is that the maximization step only returns interpretations which are maximal relative to some restriction R. Hence, some additional filtering is necessary, which is done with the help of a verifier.

Algorithm 4.25: $initializationStep^{R}_{prf}(D)$

```
1 ρ ← ⋃_{s∈A} {φ^Γ_{R,s}, φ^{pr}_{R,s}}
2 ρ ← process_{kbip}(ρ)
3 run generationStep^{R,U}_{prf}(ρ)
```

Algorithm 4.25 is identical to Algorithm 4.23, which is not surprising since the admissible and the preferred semantics share the same first steps. The algorithm initializes ρ with the conflict-free encodings and then applies the k-bipolar state processor to ensure the computation of only admissible interpretations.

This step now differs from the admissible pipeline. Algorithm 4.26 has a next step, thus a $unverifiedProcessingStep^{R,\mathcal{U}}_{prf}(I)$ call is wired into the algorithm. It also has to deal with

Algorithm 4.26: $generationStep^{R,\mathcal{U}}_{prf}(\rho)$

```
1 U' ← U
2 ρ ← ρ ∪ U'
3 U ← U \ U'
4 I ← generate^R_cf(ρ)
5 if I ≠ ∅ then
6 |   run unverifiedProcessingStep^{R,U}_prf(I)
7 |   run generationStep^{R,U}_prf(ρ)
8 end
```

the update formulas in $\mathcal{U}$, which are communicated back by the maximizers. Besides that, it exhaustively computes admissible interpretations I until $I = \emptyset$ holds.

Algorithm 4.27: $unverifiedProcessingStep^{R,\mathcal{U}}_{prf}(I)$

```
1 (J, ψ) ← maximize^R_adm(I)
2 U ← U ∪ {ψ}
3 run verificationStep^{R,I}_prf(J)
```

Algorithm 4.27 takes an admissible interpretation I and calls the restricted admissible maximizer on it. The resulting update formula is then communicated back to the previous step via the shared data structure $\mathcal{U}$. Note that the interpretation J is only maximal relative to restriction R. Hence, it remains to verify if it is also maximal in ADF D.

Algorithm 4.28: $verificationStep^{R,\mathcal{I}}_{prf}(I)$

```
1 if verify_prf(I) then
2 |   I ← I ∪ {I}
3 end
```

Algorithm 4.28 is the first step that differs from the sequential algorithm for preferred interpretations. If not dealing with restricted search spaces, there is no need for an extra verification step, since the maximizer computes preferred interpretations directly. In the restricted case it becomes necessary to filter non-maximal results.

4.6.6 Stable

The stable algorithm is again in line with its sequential counterpart. There is no conceptual difference, also no need for further building blocks.

Algorithm 4.29: $initializationStep^{R}_{stb}(D)$

1 $\rho \leftarrow \bigcup_{s \in A}\{\phi^{mod}_{R,s}\}$
2 **run** $generationStep^{R}_{stb}(\rho)$

The first step of the pipeline is the initialization step, represented by Algorithm 4.29. Since stable models are a subset of two-valued models, two-valued models are generated as candidates. This is achieved by encoding the two-valued model property to ρ.

Algorithm 4.30: $generationStep^{R}_{stb}(\rho)$

1 $I \leftarrow generate^{R}_{mod}(\rho)$
2 **if** $I \neq \emptyset$ **then**
3 | **run** $verificationStep^{R,\mathcal{I}}_{stb}(J)$
4 | **run** $generationStep^{R}_{stb}(\rho)$
5 **end**

Algorithm 4.30 exhaustively generates two-valued models. These interpretations are then handed to the verification step.

Algorithm 4.31: $verificationStep^{R,\mathcal{I}}_{stb}(I)$

1 **if** $verify_{stb}(I)$ **then**
2 | $\mathcal{I} \leftarrow \mathcal{I} \cup \{I\}$
3 **end**

Algorithm 4.31 performs the stable model verification and discards all non-stable I. It is the last step of the stable model computation, therefore all remaining I are added to $\mathcal{I}$.

4.6.7 Complete

The parallel algorithm for the complete semantics is in line with its sequential counterpart.

Algorithm 4.32: $initializationStep^{R}_{com}(D)$

1 $\rho \leftarrow \bigcup_{s \in A}\{\phi^{\Gamma}_{R,s}, \phi^{pr}_{R,s}\}$
2 $\rho \leftarrow process_{kbip}(\rho)$
3 **run** $generationStep^{R}_{com}(\rho)$

Since complete interpretations are a subset of admissible interpretations, Algorithm 4.32 applies the k-bipolar state processor to ρ. In combination with the conflict-free encodings, only admissible interpretations then satisfy ρ.

Algorithm 4.33: $generationStep^{R}_{com}(\rho)$

```
1 I ← generate^R_cf(ρ)
2 if I ≠ ∅ then
3 |   run verificationStep^{R,I}_com(J)
4 |   run generationStep^R_com(ρ)
5 end
```

Algorithm 4.33 exhaustively computes admissible interpretations and hands them to the verification step.

Algorithm 4.34: $verificationStep^{R,\mathcal{I}}_{com}(I)$

```
1 if verify_com(I) then
2 |   I ← I ∪ {I}
3 end
```

Algorithm 4.34 now discards all those interpretations that do not pass the complete verification. This is the last step of the parallel algorithm for the complete semantics, it therefore returns all remaining interpretations I to $\mathcal{I}$.

4.7 Reasoning Tasks

One interesting side effect of the restricted semantics is its usage in various reasoning tasks. Until now, we have mostly discussed the enumeration of the interpretations of some ADF D and some semantics σ. However, there are further interesting reasoning tasks to consider. In this section, we show how we can use all the discussed concepts to assemble algorithms for these reasoning tasks without much further work.

4.7.1 Credulous Reasoning

The first reasoning task to consider is credulous reasoning, it asks if some argument is true in at least one interpretation of some ADF D. Algorithm 4.35 makes only use of previously defined algorithms by reformulating this question. It asks only for those interpretations $\mathcal{I}$ of ADF D in which argument a is true by using the restriction $\{\mathbf{t}(a)\}$. If $\mathcal{I}$ is not empty, then there obviously exists some interpretation in which argument a is true. Note that it is not necessary to compute the whole set $\mathcal{I}$, one can stop after the first result. However, for illustration purposes we have reused the $enumerate_\sigma$ algorithm here.

Proposition 20 *Let $D = (A, L, C)$ be some ADF and $a \in A$ some argument, then $\exists I \in \sigma(D) : I(a) = \mathbf{t}$ iff $cred_\sigma(D, a) = true$.*

Algorithm 4.35: $cred_\sigma(D, a)$

```
1 I ← enumerate_σ^{t(a)}(D)
2 return I ≠ ∅
```

Proof. Assume $\exists I \in \sigma(D) : I(a) = \mathbf{t}$, then $enumerate_\sigma^{\{\mathbf{t}(a)\}}(D)$ computes I at one point. Hence, it holds $I \in \mathcal{I}$ and further $\mathcal{I} \neq \emptyset$ from which $cred_\sigma(D, a) = true$ follows.

Assume $cred_\sigma(D, a) = true$, then $\mathcal{I} \neq \emptyset$. Since $enumerate_\sigma^{\{\mathbf{t}(a)\}}(D)$ only computes interpretations I satisfying the restriction $\{\mathbf{t}(a)\} \subseteq I$ it holds $\exists I \in \sigma(D) : I(a) = \mathbf{t}$. □

4.7.2 Skeptical Reasoning

Another important reasoning task is skeptical reasoning, it asks if some argument is true for all interpretations of some ADF D. Although the question is quite similar to credulous reasoning, the algorithm is a bit trickier. Algorithm 4.36 splits this question into two sub-questions, first it asks if there exists some interpretation I with $I(a) = \mathbf{f}$ then it asks if there exists some interpretation I with $I(a) = \mathbf{u}$. If one of these sub-questions is answered positively, we know that $I(a) = \mathbf{t}$ cannot hold for all interpretations, hence the algorithm returns false. If none of these sub-questions can be answered positively, then we have found no counterexample to $I(a) = \mathbf{t}$, hence it must hold for all interpretations of D. Note again that we use $enumerate_\sigma$ mainly for simplicity, we do not have to compute the whole set $\mathcal{I}$ to answer if $\mathcal{I} \neq \emptyset$ respectively $\mathcal{I} = \emptyset$ holds.

Algorithm 4.36: $skept_\sigma(D, a)$

```
1 I ← enumerate_σ^{f(a)}(D)
2 if I ≠ ∅ then
3 |   return false
4 end
5 I ← enumerate_σ^{u(a)}(D)
6 return I = ∅
```

Proposition 21 *Let $D = (A, L, C)$ be some ADF and $a \in A$ some argument, then $\forall I \in \sigma(D) : I(a) = \mathbf{t}$ iff $skept_\sigma(D, a) = true$.*

Proof. Assume $\forall I \in \sigma(D) : I(a) = \mathbf{t}$, then $enumerate_\sigma^{\{\mathbf{f}(a)\}}(D) = \emptyset$ thus we continue with the second sub-question. It further holds $enumerate_\sigma^{\{\mathbf{u}(a)\}}(D) = \emptyset$ and therefore $\mathcal{I} = \emptyset$, which results in $skept_\sigma(D, a) = true$.

Assume $skept_\sigma(D, a) = true$, then $enumerate_\sigma^{\{\mathbf{f}(a)\}}(D) = \emptyset$ follows, since otherwise we would enter the if-statement which would result in a contradiction. It further holds

$enumerate_\sigma^{\{\mathbf{u}(a)\}}(D) = \emptyset$ since $\mathcal{I} = \emptyset$ must hold in order to return true. Hence, since both sub-questions are not able to provide a counterexample, $\forall I \in \sigma(D) : I(a) = \mathbf{t}$ holds. □

4.7.3 Verification

The verification reasoning task must not be confused with the verifier building block. Although, at first glance it seems similar, the reasoning task is more comprehensive. It asks whether I is some interpretation of ADF D under semantics σ, i.e. $I \in \sigma(D)$. While the verifier building block just verifies a single property of an interpretation. Take the preferred verifier for instance, which expects a given interpretation to be admissible, but does not check it by itself. Assume some non-admissible interpretation I, in order to check $I \in prf(D)$ we cannot just use the preferred verifier, because it only checks for some interpretation J s.t. $I <_i J$. If it does not find such a J, then we would assume that $I \in prf(D)$ holds, although I is not even admissible.

However, we already have everything we need to solve this reasoning task. We again make use of the existing $enumerate_\sigma$ algorithm under some restriction. If the computation is restricted to I, then only I itself can satisfy the restriction $I \subseteq I$. Hence, it then either holds $\mathcal{I} = \{I\}$ or $\mathcal{I} = \emptyset$ and the reasoning task is reduced to a single $\mathcal{I} \neq \emptyset$ check.

Algorithm 4.37: $ver_\sigma(D, I)$

```
1 I ← enumerate_σ^I(D)
2 return I ≠ ∅
```

Proposition 22 *Let $D = (A, L, C)$ be some ADF and I some interpretation of D, then $I \in \sigma(D)$ iff $ver_\sigma(D, I) = true$.*

Proof. Assume $I \in \sigma(D)$, then $enumerate_\sigma^I(D) = \{I\}$ and therefore $\mathcal{I} \neq \emptyset$, hence $ver_\sigma(D, I) = true$.

Assume $ver_\sigma(D, I) = true$, then $\mathcal{I} \neq \emptyset$ and therefore $I \in \mathcal{I}$, hence $I \in \sigma(D)$. □

4.7.4 Parallelization

It is possible to decide the skeptical and the credulous reasoning problems in parallel. This is illustrated for the credulous reasoning problem, but it works for the skeptical reasoning problem analogously.

First observe Algorithm 4.38, which makes use of the already known restriction mechanism. It behaves exactly the same as Algorithm 4.35 but tries to answer a slightly modified question, namely whether $\exists I \in \sigma(D) : R \subseteq I \wedge I(a) = \mathbf{t}$ holds. Note that in order to avoid an inconsistent R' it is required that $R \cap \{\mathbf{f}(a), \mathbf{u}(a)\} = \emptyset$ holds.

Algorithm 4.38: $cred_\sigma^R(D, a)$

```
1 R' ← R ∪ {t(a)}
2 I ← enumerate_σ^{R'}(D)
3 return I ≠ ∅
```

Proposition 23 *Let $\mathcal{R} = \{R_1, \ldots, R_n\}$ be a set of restrictions covering the whole search space and let $R_i \cap \{\mathbf{f}(a), \mathbf{u}(a)\} = \emptyset$ hold for all $R_i \in \mathcal{R}$. Then it holds $cred_\sigma(D, a) = false$ if $cred_\sigma^{R_i}(D, a) = false$ for all $R_i \in \mathcal{R}$ and $cred_\sigma(D, a) = true$ if $cred_\sigma^{R_i}(D, a) = true$ for some $R_i \in \mathcal{R}$.*

Proof. Assume $cred_\sigma^{R_i}(D, a) = false$ for all $R_i \in \mathcal{R}$, then $enumerate_\sigma^{R_i \cup \{\mathbf{t}(a)\}}(D) = \emptyset$ for all $R_i \in \mathcal{R}$. Observe that since $\mathcal{R}$ covers the whole search space it holds $\bigcup_{R_i \in \mathcal{R}} enumerate_\sigma^{R_i \cup \{\mathbf{t}(a)\}}(D) = enumerate_\sigma^{\{\mathbf{t}(a)\}}(D) = \emptyset$ and therefore $cred_\sigma(D, a) = false$.

Assume $cred_\sigma^{R_i}(D, a) = true$ for some $R_i \in \mathcal{R}$, then there exists some interpretation $I \in \sigma(D)$ with $I(a) = \mathbf{t}$. It further holds $I \in enumerate_\sigma^{\{\mathbf{t}(a)\}}(D)$ and therefore $cred_\sigma(D, a) = true$. □

Algorithm 4.39: $cred_\sigma(D, a)$

```
1 res ← false
2 R ← decompose_3(D, {a})
3 foreach restriction R ∈ R do
4 |   run res ← res ∨ cred_σ^R(D, a)
5 end
6 return res
```

Algorithm 4.39 is the result of the combination of Proposition 23 with Algorithm 4.38. The idea is to decompose the search space and then decide the credulous reasoning problem for distinct parts of the search space in parallel. Some shared and synchronized *res* variable is used, which is rendered true if at least one part of the search space contains an interpretation I with $I(a) = \mathbf{t}$. The algorithm can stop immediately as soon as *res* becomes true. Note that Algorithm 4.39 is kept simple to illustrate the overall scheme, it is in no sense an optimized version. In practice there is some bookkeeping necessary to determine if all tasks are done or to cancel running tasks if one of them already returned true.

CHAPTER 5

Experiments

The proposed algorithms were implemented as a part of the TweetyProject, which is a comprehensive collection of Java libraries for logical aspects of artificial intelligence [Thi14]. Hence, our system was written in Java, which has advantages and disadvantages. The system was designed as a library and Java is still one of the most dominant programming languages, there is therefore a huge potential audience. The further improvement and maintenance of the system is an important and necessary step from academia towards the industry. However, one of the drawbacks of the Java implementation is the higher level of abstraction, compared to C/C++ implementations. Although it interacts with native SAT solvers via the Java Native Interface (JNI) [Lia99] there is some non-negligible overhead imposed by JNI itself and some mapping layer between the SAT solver binding and the ADF reasoner. It is possible to get rid of this mapping layer by a tighter SAT solver integration, there are however some engineering problems to solve first. Hence, future improvements of the system are trying to minimize the gap to native implementations.

The goal of this chapter is to provide some proof-of-concept for the parallelization approach. Hence, since the implemented system is capable of performing all reasoning tasks in sequential and in parallel, the experiments focus on the difference of these execution models. This way we do not measure technological differences, since both models have the same foundation, but the pure impact of parallelization on the runtime.

5.1 Java Microbenchmark Harness (JMH)

Many optimizations are not performed by the Java compiler, but during runtime by the Java Virtual Machine (JVM). Hence, benchmark results may differ, depending on how the JVM executes some code passages. Execution varies from interpretation to just-in-time compiled highly optimized code incorporating collected runtime data. To give the JVM a chance to collect runtime data and perform optimizations based on them, some warmup iterations are necessary before collecting benchmark results. However,

warmup may not be sufficient, the JVM is a highly complex piece of software and there is a lot one can do wrong. Fortunately, there is something called Java Microbenchmark Harness (`https://github.com/openjdk/jmh`) which deals with all the JVM magic and makes it easier to write correct microbenchmarks. JMH is the defacto standard when dealing with microbenchmarks on the JVM and is therefore used by the following experiments.

As a configuration the number of forks is set to 3, the number of warmup iterations set to 5 and the number of measured iterations is also set to 5. Each fork starts a new Java process, which then runs the benchmarks. It is useful to neglect run-to-run variance. As a side-effect the collected runtime profiles are lost between each fork. Hence, in total we run each iteration 30 times, 10 times for each fork, but only 5 of them are measured. Each iteration takes at least 10 seconds and runs the benchmark at least once. If a benchmark runs multiple times within these 10 seconds, then the average time is taken. This ensures more stable results, since environmental influences have more impact on benchmarks with only short running times.

5.2 Setup

It is not expected that parallelization is useful for each instance or semantics. This is best illustrated by the grounded semantics. If just a few interpretations are generated per instance while the semantics has no building blocks besides the generator, then the parallelization overhead may be counterproductive. Therefore the experiment first focuses on a few selected instances purely chosen by their number of interpretations. A high number of interpretations may also indicate that more parts of the search space are relevant, which is beneficial for the decomposition. As a follow-up we perform further experiments to see if the findings are generalizable or the selected instances are outliers.

The following instances can be found at `https://www.dbai.tuwien.ac.at/proj/adf/yadf/`:

1. adfgen_nacyc_se05_a_02_s_02_b_02_t_02_x_02_c_sXOR_Traffic_benton-or-us.gml.80_25_56.apx.adf
2. adfgen_nacyc_se05_a_02_s_02_b_02_t_02_x_02_c_sXOR_ABA2AF_afinput_exp_acyclic_depvary_step5_batch_yyy06_29_57.apx.adf
3. adfgen_nacyc_se05_a_02_s_02_b_02_t_02_x_02_c_sXOR_ABA2AF_afinput_exp_acyclic_indvary1_step6_batch_yyy08_35_52.apx.adf

Table 5.1 shows the number of interpretations per instance and semantics. The semantics naive, admissible and complete were chosen. Although the complete semantics does not have many interpretations, it still generates the admissible interpretations and additionally applies a verification step.

	naive	admissible	complete
Instance 1	104085	39059	372
Instance 2	72765	86272	8
Instance 3	16404	131072	1

Table 5.1: Number of interpretations per instance and semantics.

The experiments run on an AMD Ryzen 7 3700X 8-Core processor with 16 logical cores. We run it in three different modes, as a baseline the interpretations are sequentially computed then we perform two parallel runs. The first run uses the Most Complex Acceptance Condition (MCA) decomposition heuristics. The second run uses the Least Complex Acceptance Condition (LCA) decomposition heuristics. Both heuristics select 3 arguments, which results in 27 search space decompositions. The number of arguments is determined by the 16 logical cores, since 2 arguments with 9 decompositions would underutilize the processor. Although the system has also bindings for Lingeling [Bie17] and Picosat [Bie08], MiniSat [ES04] was chosen as an underlying SAT solver.

5.3 Results

Table 5.2 shows the average run time and standard error of each experiment. Although the concurrent execution model is less deterministic than its sequential counterpart, the results are quite stable with just a few outliers like the Naive LCA run.

In the following we illustrate the parallel results relative to the sequential results, which functions as a baseline. This makes it easy to determine the speedup factor in comparison to the traditional sequential execution.

Figure 5.1 shows the relative results for the admissible semantics run. One can already see, especially by looking at Instance 3, the importance of the search space decomposition for the parallel execution. While for Instance 1 the MCA heuristics outperforms the LCA heuristics, for Instance 2 and Instance 3 the opposite is the case. The sequential

		Instance 1	Instance 2	Instance 3
Admissible	**Seq.**	1,491 ± 0,033	6,021 ± 0,344	12,344 ± 0,461
	LCA	0,294 ± 0,030	0,699 ± 0,074	1,048 ± 0,064
	MCA	0,170 ± 0,008	2,572 ± 0,072	11,833 ± 0,853
Complete	**Seq.**	6,972 ± 0,155	31,729 ± 0,683	48,567 ± 1,383
	LCA	1,091 ± 0,036	3,985 ± 0,151	5,639 ± 0,070
	MCA	1,109 ± 0,026	5,725 ± 0,702	17,163 ± 0,991
Naive	**Seq.**	30,123 ± 0,316	37,482 ± 0,626	8,604 ± 0,224
	LCA	26,846 ± 7,771	256,015 ± 35,065	26,726 ± 0,592
	MCA	9,405 ± 0,193	11,840 ± 0,550	6,278 ± 0,176

Table 5.2: Raw benchmark data in seconds.

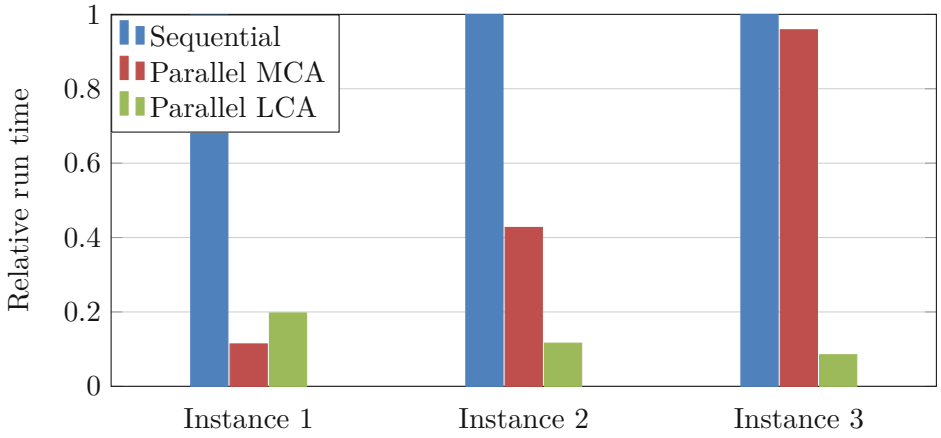


Figure 5.1: Relative run time of the admissible semantics.

execution is almost on par with the parallel MCA execution for Instance 3. This can happen since the interpretations are usually not uniformly distributed over all search space partitions, resulting in many exhausted partitions at an early computation stage. The remaining partitions then underutilize the processor.

Figure 5.2 shows the relative results for the complete semantics. The complete verifier is stateless, allowing for a good hardware utilization even if the decomposition of the search space is not ideal. This also explains the better result of the MCA heuristics on Instance 3 compared to the admissible semantics. The results also indicate that the verification is the bottleneck of the computation, otherwise the bar charts would resemble the ones of the admissible semantics. This conclusion can also be drawn from the raw benchmark data, since the complete semantics takes way longer to compute although it only differs in the additional verification step from the admissible semantics.

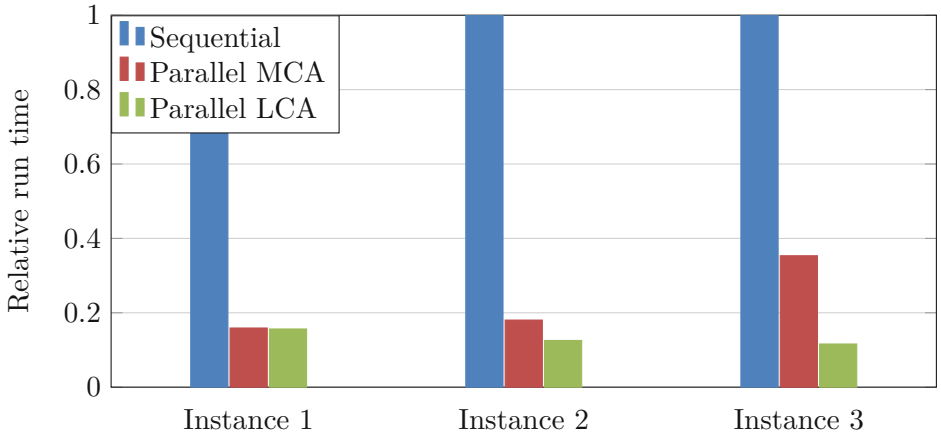


Figure 5.2: Relative run time of the complete semantics.

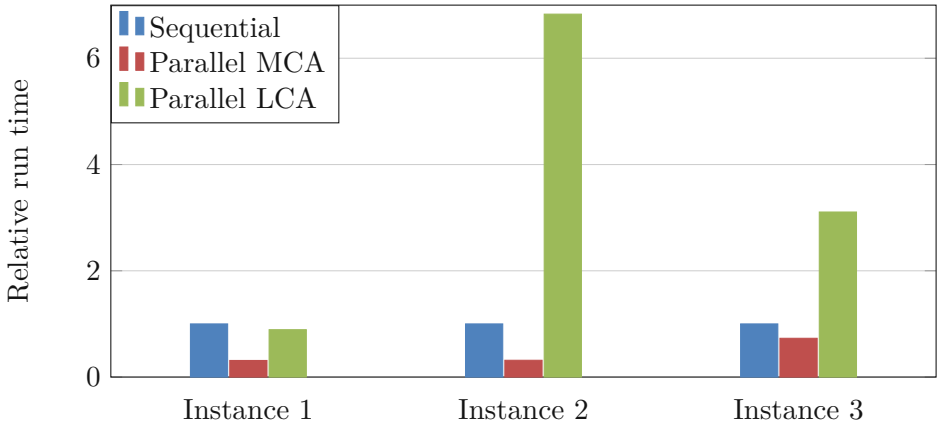


Figure 5.3: Relative run time of the naive semantics.

Figure 5.3 shows the relative results for the naive semantics. There is something unexpected but interesting happening for Instance 2 and Instance 3. Sometimes it has to be expected that the additional parallelization overhead leads to worse results compared to the sequential execution. This is however not the case here, since the additional overhead usually becomes insignificant if an instance takes long enough to compute.

Recall the role of the interpretation processor for the naive semantics, which takes an interpretation and maximizes it according to the information ordering $<_i$. The maximization process is not one-to-one, hence there are two different interpretations I_1 and I_2 with $\mathit{maximize}(I_1) = \mathit{maximize}(I_2)$. If the computation happens sequentially and I_1 is computed first, then the computation of I_2 is ruled out by the maximizer. If the computation however happens not sequentially and I_1 and I_2 lie in different partitions of the search space, then both are computed. Interpretations can only be ruled out by other interpretations within the same partition, since the current execution model does not share any information between different partitions. Therefore, for semantics with an interpretation processor, redundant computation can happen.

Another effect also takes place, the implementation of the interpretation processor is stateless, the naive verifier is stateful. This allows the execution framework to spawn and execute arbitrarily many maximization tasks per partition, but the execution of only one verification per partition at a time. This combination relies too much on the scheduling mechanism and explains the high standard error of the runtime for these two instances. If the scheduler does not balance the maximization tasks and the verification tasks, then the system has to deal with back pressure, which results in less throughput. One simple solution is the use of a stateless verifier, which removes the restriction of one verification at a time and branch. However, sometimes it is beneficial to rely on statefulness, the naive verifier saves some encoding time for instance. A more advanced and probably better solution is a scheduling mechanism which takes runtime characteristics into account to

	Sequential	Parallel LCA	Parallel MCA
Admissible	224,274 ± 14,846	20,367 ± 5,276	105,712 ± 25,342
Complete	770,388 ± 54,758	81,825 ± 19,514	139,856 ± 40,132

Table 5.3: Accumulated results of the follow-up runs.

avoid back pressure because of potential bottlenecks.

5.4 Follow-Up

A subset of another 30 instances was picked from the YADF page and benchmarked with the admissible and the complete semantics. All of these instances were solvable within 10 seconds by one of the parallel runs. The admissible semantics was chosen to further illustrate the importance of the right search space decomposition heuristics. The complete semantics was chosen to show that a bad search space decomposition can be mitigated by the semantics decomposition. Further experiments with the preferred semantics are postponed to a later date, since we already gathered enough insight for necessary future improvements before follow-up experiments are useful.

This time each experiment was only run three times, Table 5.3 shows the average runtime and the error. Figure 5.4 shows the relative speedup in comparison to the sequential run.

What can be concluded from these results is that the LCA heuristics performs much better for the admissible and complete semantics. It also shows the importance of both decomposition approaches and that the previous results were not just outliers, but that parallelization often leads to performance improvements. More extensive experiments are performed in the future, once the current problems are solved and the system is in a more mature state.

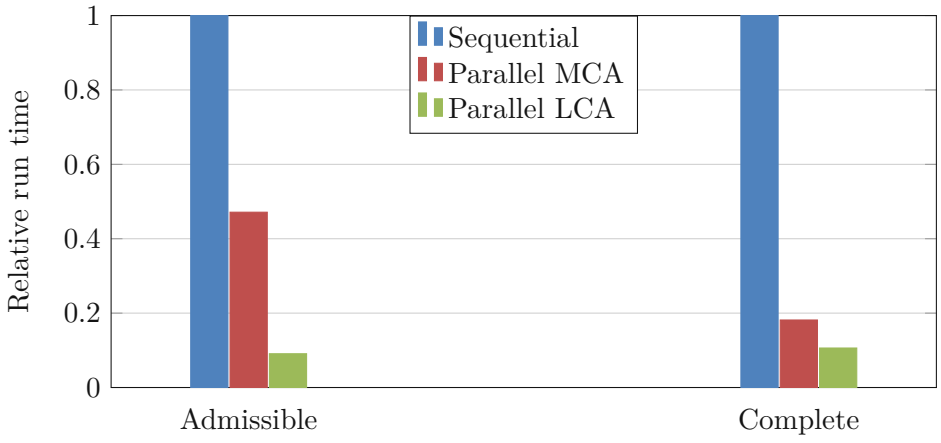


Figure 5.4: Relative accumulated run time of the follow-up runs.

CHAPTER 6

Related Work

One of the starting points of this thesis was the paper [LMN+18]. It provides a basic framework for SAT based ADF systems and also advanced encodings on which our system heavily relies on like the k-bipolar optimizations. It also resulted in the k++adf system, which functions as a main motivation for the parallelization approach. An ongoing goal is to outperform k++adf for most instances, which is hard to achieve with classical approaches because of technical constraints imposed by Java. Therefore we started investigating and implementing parallelization approaches.

A different approach to our SAT based system is illustrated by the systems YADF [BDH+20] and DIAMOND [ES14] which both rely on Answer Set Programming (ASP) [EIK09] as their target formalism. A system which relies on Quantified Boolean Formulas (QBFs) [SBPS19] as its target formalism was introduced in [DWW15]. The additional expressiveness of ASP or QBFs may be tempting at first, but is not necessary for parallelization. It is undesirable for the complexity of the building blocks to exceed the borders of NP. In fact it is desirable to have many easy to compute building blocks, since these are the atomic objects which we can run in parallel. Hence, approaches with just a single ASP or QBF encoding would prevent the current parallelization framework to run things in parallel.

None of these systems currently make use of some form of parallelization. There is some work on parallel algorithms for other argumentation formalisms [CTV+15], but it often only involves a certain semantics and is further not directly applicable to ADFs.

CHAPTER 7

Outlook and Conclusion

It was shown that parallelization of ADF algorithms is a viable approach that opens a new path for further research. The experiments have indicated that the key to beneficial parallelization is the search space decomposition. Therefore, further work has to be spent on better decomposition heuristics. However, even the best heuristics can not determine how the interpretations are distributed over the search space. Therefore, a plausible approach to investigate is the dynamic decomposition during runtime. If one partition of the search space is exhausted, it could be viable to further decompose a not yet exhausted partition. This would make the choice of the right decomposition heuristics less crucial. It would also lead to a less volatile hardware utilization. Although the problem is easily formulated, this is some hard engineering problem and requires a carefully designed system.

Another encountered problem deals with the main result of [LMN+18]. The proposed k-bipolar optimizations are too beneficial to skip them entirely. However, if the number of dependent links becomes too high, most of the reasoning time is spent on the computation of the clauses and the interaction with the underlying SAT solver. It is planned to improve our system further, by reducing the current interaction overhead with the SAT solver. There are instances which are computable by k++adf but not by our system, at least not in reasonable time. Work in this direction should further close the gap between these two systems. However, at one point it becomes unfeasible to compute the k-bipolar encodings a priori. Hence, some investigation on dynamically computed encodings based on the runtime profile may be tempting, in order to solve instances which are currently too hard for either system.

The implemented library currently supports 8 different semantics, namely conflict-free, naive, admissible, preferred, complete, two-valued, stable and grounded. For each of these semantics it is possible to spend a significant amount of time in search for improved algorithms. Hence, it is expected that future improvement not only happens by finding more efficient execution models, but by improved sequential algorithms.

From an engineering standpoint there is still potential for future developments. One can investigate different queueing and scheduling mechanisms for the concurrent execution of the building blocks. The current implementation has one global queue and one thread pool. It is a possibility to have multiple queues and thread pools. For instance, each partition of the search space can have its own queue and thread pool. Future experiments can also deal with advanced scheduling mechanisms based on priorities or collected runtime data. It is currently not known if a system could benefit from research in this area. There are also interesting developments happening in the Java world, like Project Loom (`https://openjdk.java.net/projects/loom/`) or Project Panama (`https://openjdk.java.net/projects/panama/`). The former may help with parallelization and the latter with a better SAT solver integration.

It is also important to underline the library aspect of the proposed system. Abstract Dialectical Frameworks function as a target formalism for other research, it is therefore from importance to have a library in a widespread language like Java. There is also work on the application of ADFs in the industry, especially in law, as [AAABC16] shows. A well-designed library can be a helpful step in this direction. However, there is still a lot to work on, like documentations, user guides, easy-to-use APIs, tests, extensibility and maybe frontends as an abstraction over complicated details.

List of Figures

List of Tables

List of Algorithms

Bibliography

[AAABC16] Latifa Al-Abdulkarim, Katie Atkinson, and Trevor Bench-Capon. A methodology for designing systems to reason with legal cases using abstract dialectical frameworks. *Artificial Intelligence and Law*, 24:1–49, 03 2016.

[BCD07] Trevor J. M. Bench-Capon and Paul E. Dunne. Argumentation in artificial intelligence. *Artificial Intelligence*, 171(10):619–641, 2007.

[BCG11] Pietro Baroni, Martin Caminada, and Massimiliano Giacomin. An introduction to argumentation semantics. *Knowledge Engineering Review*, 26:365–410, 12 2011.

[BDH+20] Gerhard Brewka, Martin Diller, Georg Heissenberger, Thomas Linsbichler, and Stefan Woltran. Solving advanced argumentation problems with answer set programming. *Theory and Practice of Logic Programming*, 20(3):391–431, Jan 2020.

[BDW11] Gerd Brewka, Paul E. Dunne, and Stefan Woltran. Relating the semantics of abstract dialectical frameworks and standard AFs. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two*, IJCAI'11, page 780–785. AAAI Press, 2011.

[BES+17] Gerhard Brewka, Stefan Ellmauthaler, Hannes Strass, Johannes Peter Wallner, and Stefan Woltran. Abstract dialectical frameworks. an overview. *IfCoLog Journal of Logics and their Applications*, 4(8):2263–2317, October 2017.

[Bie08] Armin Biere. Picosat essentials. *JSAT*, 4:75–97, 05 2008.

[Bie17] Armin Biere. Cadical, lingeling, plingeling, treengeling and yalsat entering the sat competition 2018. *Proceedings of SAT Competition*, pages 14–15, 2017.

[BW10] Gerhard Brewka and Stefan Woltran. Abstract Dialectical Frameworks. In Fangzhen Lin, Ulrike Sattler, and Miroslaw Truszczynski, editors, *Proceedings of the Twelfth International Conference on Principles of Knowledge Representation and Reasoning, KR 2010*, pages 102–111, Toronto, Ontario, Canada, May 2010. AAAI Press.

[CDG+15] Günther Charwat, Wolfgang Dvořák, Sarah A. Gaggl, Johannes P. Wallner, and Stefan Woltran. Methods for solving reasoning problems in abstract argumentation – a survey. *Artificial Intelligence*, 220:28–63, 2015.

[CTV+15] Federico Cerutti, Ilias Tachmazidis, Mauro Vallati, Sotirios Batsakis, Massimiliano Giacomin, and Grigoris Antoniou. Exploiting parallelism for hard problems in abstract argumentation. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, page 1475–1481. AAAI Press, 2015.

[DD17] Wolfgang Dvorák and E. Paul Dunne. Computational problems in formal argumentation and their complexity. *IfCoLog Journal of Logics and their Applications*, 4, 2017.

[Dun95] Phan Minh Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artif. Intell.*, 77(2):321–357, September 1995.

[DWW15] Martin Diller, Johannes Peter Wallner, and Stefan Woltran. Reasoning in abstract dialectical frameworks using quantified boolean formulas. *Argument & Computation*, 6(2):149–177, 2015.

[DZLW20] Martin Diller, Atefeh Zafarghandi, Thomas Linsbichler, and Stefan Woltran. Investigating subclasses of abstract dialectical frameworks. *Argument & Computation*, 11:1–29, 01 2020.

[EIK09] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer set programming: A primer. In *Reasoning Web. Semantic Technologies for Information Systems: 5th International Summer School 2009, Brixen-Bressanone, Italy, August 30 - September 4, 2009, Tutorial Lectures*, page 40–110. Springer-Verlag, Berlin, Heidelberg, 2009.

[ES04] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, pages 502–518, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[ES14] Stefan Ellmauthaler and Hannes Strass. The diamond system for computing with abstract dialectical frameworks. *Computational Models of Argument-Proceedings of COMMA 2014. Frontiers in Artificial Intelligence and Applications*, 266:233–240, 01 2014.

[Lia99] Sheng Liang. Java native interface: Programmer's guide and specification. 1999.

[LMN+18] Thomas Linsbichler, Marco Maratea, Andreas Niskanen, Johannes P. Wallner, and Stefan Woltran. Novel algorithms for abstract dialectical frameworks based on complexity analysis of subclasses and sat solving. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, IJCAI'18, page 1905–1911. AAAI Press, 2018.

[PGB+05] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

[Pra18] Henry Prakken. Historical overview of formal argumentation. In Pietro Baroni, Dov Gabbay, Massimiliano Giacomin, and Leendert van der Torre, editors, *Handbook of Formal Argumentation*, volume 1, pages 73–141. College Publications, 2018.

[PWW13] Sylwia Polberg, Johannes Peter Wallner, and Stefan Woltran. Admissibility in the abstract dialectical framework. In *Proceedings of the 14th International Workshop on Computational Logic in Multi-Agent Systems - Volume 8143*, CLIMA XIV, page 102–118, Berlin, Heidelberg, 2013. Springer-Verlag.

[SBPS19] Ankit Shukla, Armin Biere, Luca Pulina, and Martina Seidl. A survey on applications of quantified boolean formulas. In *31st IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2019, Portland, OR, USA, November 4-6, 2019*, pages 78–84. IEEE, 2019.

[Thi14] Matthias Thimm. Tweety: A comprehensive collection of java libraries for logical aspects of artificial intelligence and knowledge representation. In *Proceedings of the Fourteenth International Conference on Principles of Knowledge Representation and Reasoning*, KR'14, page 528–537. AAAI Press, 2014.