11-2023

Using metaprogramming techniques to enhance eclingo performance through the reification format

Eleuterio Juan Lillo Portero

Using metaprogramming techniques to enhance *eclingo* performance through the reification format.

A Thesis

Presented to the

Department of Computer Science

and the

Faculty of the Graduate College

University of Nebraska

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

University of Nebraska at Omaha

by

Eleuterio Juan Lillo Portero

November 2023

Supervisory Committee

Dr. Jorge Fandinno

Dr. Yuliya Lierler

Dr. Ada-Rhodes Short

Using metaprogramming techniques to enhance *eclingo* performance through the reification format.

Eleuterio Juan Lillo Portero, MS

University of Nebraska, 2023

Advisor: Dr. Jorge Fandinno

Answer Set Programming is an automated reasoning technology that has become a prime candidate for solving knowledge-intense search and optimization problems. One of the main reasons of its success is the availability of highly effective solvers that can go toe-to-toe with Satisfiability Solvers while dealing with a high-level human understandable language. Epistemic logic programs are an extension of Answer Set Programming with subjective literals that allow to succinctly represent several problems that cannot be represented using the standard language of Answer Set Programming. `eclingo` is a solver developed to solve problems described in the language of Epistemic Logic Programs. This research aims to enhance the efficiency of such solver. The focus of the research will be aimed at the use of the metaprogramming capabilities of Answer Set Programming solver `clingo`. This will allow us to enhance the solver with new inference rules expressed in the Answer Set Programming language. This will reduce the search space and, in principle, improve solver performance.

Acknowledgement

I would like to first thank my thesis committee members Dr. Yuliya Lierler, Dr. Jorge Fandinno, and Dr. Ada-Rhodes Short. In particular I would like to thank Dr Fandinno. He opened the doors of research for me 2 summers ago, introduced me to ASP and eclingo and spent several hours explaining and helping me debug this project. Without that help I would not have been able to finish this project.

Contents

Figures

Tables

Algorithms

Acronyms

AST Abstract Syntax Tree

ASP Answer Set Programming

KRR Knowledge Representation and Reasoning

SAT Satisfiability

ELP Epistemic Logic Program

C19 Epistemic Logic Semantics: Cabalar 2019

K15 Epistemic Logic Semantics: Kahl 2019

G94 Epistemic Logic Semantics: Gelfond 1994

BT Bomb in the Toilet Benchmark

Glossary

$At(\prod)$ is the set of all of the atoms that happen in the epistemic program $\prod$. 2

$Facts(\prod)$ is the set of atoms that occur as facts in $\prod$. 2

$Heads(\prod)$ is the set of all the atoms that occur in the head (left-hand side) of any rule. 2

W is the set of interpretations of an epistemic logic program. 2

$\prod$ is an epistemic program. 2

$\varphi(P)$ is the grounded version of a program where all ground terms are ground terms. 1

answer set programming (ASP) is a form of declarative programming used in the modelling of hard combinatorial problems and in knowledge representation and reasoning. 1

atom or **atomic formula** is a formula that has no strict sub-formulas. 2

clingo is the ASP solver created by Potassco which underlies eclingo. 2

eclingo is the Epistemic Logic Program solver. 2

fact is a predicate that always takes the truth value of True.. 12

formula is a finite sequence of symbols from a given alphabet that is part of a formal language. 9

ground terms are terms that do not contain variables. 1

predicate is a statement or mathematical assertion that contains variables, and may be true, false or unknown depending on those variables' values.. 9

reification is the process of transforming a (ground) logic program into a set of facts. 12

stable models is a set of atoms that constitutes a minimal supported model of a program $\prod$. 2

subjective literal is an expression of the form $\mathbf{K}l$ and $\mathbf{M}l$ where l is an *objective literal*, either an atom q, its explicit negation -q, or any of these preceded by the default negation represented by *not*. 2

term mathematical object that serves as component of a formula. 9

world views sets of stable models of a given program $\prod$. 2

1 Introduction

Epistemology, a term derived from the Greek epistēmē ("knowledge") and logos ("reason"), is the study of knowledge. It was the Greek philosopher Aristotle on which some of his texts first described and set the origins for discussions of the logic of knowledge and belief. Epistemic logic is a sub-field of epistemology involved in the logical approaches to knowledge and agents' beliefs. This logic evolves for epistemic reasoning which is an important feature for any agent to be considered intelligent. One of the many fields where epistemic reasoning has been applied is Knowledge Representation and Reasoning (KRR) and one of the prime candidates for KRR is Answer Set Programming [1]. answer set programming (ASP) is a form of declarative programming used in the modelling of hard combinatorial problems and in knowledge representation and reasoning. ASP is based on the stable model semantics for logic programs [2]. Within ASP, programs are solved by computing stable models, and these are used to perform the search.

We consider normal logic *programs*, which are sets of *rules* of the form:

$$a_0 \leftarrow a_1,\ ...,\ a_m, not\ a_{m+1},\ ...\ , not\ a_n \tag{1a}$$

where $n \geq m \geq 0$; a_0 is propositional atom or symbol $\bot$; and a_1,...,a_n are propositional atoms (propositional symbols). A propositional logic program is a finite set of this rules. In this case, a_0 is the head and all of the atoms a_1, ..., $a_m, not\, a_{m+1}$, ..., $not\, a_n$ from the right hand side of the equation 1a are the body and therefore the justification to derive the head of such rule. It is important to mention that *not* is a modality for non-derivable known as default negation or weak negation which implies that the atom preceded by it is "*not known*" or "*not believed to be true*", while the explicit or strong negation denoted by the symbol $\neg$ allows us to distinguish between having no justification for an atom a_0, expressed by *not* a_0, and having one for the negation of a_0, expressed by $\neg a_0$. In program rules, $\neg$ can only appear in front of atoms [3]. For example:

$$\begin{aligned} &a \\ &b \leftarrow a,\ c \\ &c \leftarrow not\ d \end{aligned} \tag{2a}$$

When delving into the complexity aspects of Answer Set Programming (ASP), the determination of whether a given ASP program possesses an answer set is proven to be NP-complete [4]. This computational complexity aligns ASP with the classical propositional satisfiability problem (SAT), underlining the intricacies involved in discerning the existence of answer sets. Notably, the introduction of predicates plays a pivotal role in transforming ASP into a powerful and effective problem-solving language. Predicates serve as essential constructs that allow for a more expressive representation of knowledge and problem domains. To formalize this transition, the semantics of a predicate program denoted as P is articulated in terms of its $\varphi(P)$, denoted as $\varphi(P)$. The program $\varphi(P)$ is meticulously crafted to encompass all conceivable ground terms instantiations of rules within P, accommodating a diverse array of instances. This involves considering the specific constants present in the program and generating all possible combinations in adherence to the defined rules. In essence, the grounding

process expands the scope of ASP, enhancing its applicability and enabling a more nuanced exploration of problem-solving scenarios through the incorporation of predicates and their corresponding groundings.

Epistemic logic programs (ELPs) are an extension of answer set programming where we are allowed to use *subjective literals* in the body of rules [5, 6]. A *subjective literal* is an expression of the form $\mathbf{K}l$ and $\mathbf{M}l$ where l is an *objective literal*, either an atom q, its explicit negation -q, or any of these preceded by the default negation represented by *not*. Intuitively, subjective literal $\mathbf{K}l$ means that we know that l is true in every stable model, while $\mathbf{M}l$ means that l is true in some stable model i.e. (non-empty) sets of sets of atoms. An epistemic logic program (or epistemic specification) is a set of rules that for a given an epistemic program $\prod$, we define $At(\prod)(\prod)$ as the set of all of the atoms that happen in the epistemic program $\prod$. By definition, by $Heads(\prod)(\prod)$, we the denote the set of all the atoms that occur in the head (left-hand side) of any rule in $\prod$ and, by $Facts(\prod)(\prod)$, we denote the set of atoms that occur as facts in $\prod$. Note that $Facts(\prod) \subseteq Head(\prod) \subseteq At(\prod)$. To obtain the subjective reduct of the epistemic specifications to yield the world views, we first define W to be a set of interpretations. We write $W \mid = Kl$ if the objective literal l holds (under the usual meaning) in all of the interpretations of W *and* $W \mid = not\ Kl$ if l does not hold in some interpretation of W [7]. Then by definition, the subjective reduct of an $\prod\ \prod$ w.r.t. a set of propositional interpretations previously defined as W and expressed as $\prod^W$, is obtained by replacing each and all of the subjective literals L by the truth value $\top$ if the set of interpretations $W \mid = L$ and by the false truth value $\bot$ otherwise. Therefore, the reduct will not contain subjective literals, and will be treated as a standard logic program, thus why `eclingo` is developed as an extension of the original ASP solver `clingo`. Finally, we can express its 'epistemic' answer sets $AS[\prod^W]$, and state that the complete set of propositional interpretations defined as W is what we call a world view of an epistemic program $\prod$ only if the interpretations $W = AS[\prod^W]$. While the complexity for a classical ASP problem is NP-Complete, the complexity to decided if an epistemic logic program has a solution or world view is $\sum_3^P$ [8].

These modal operators, $\mathbf{K}l$ and $\mathbf{M}l$, were created to provide bigger introspective reasoning capabilities, particularly when reasoning with incomplete information when modeling knowledge about the world or an agent. As defined previously, the semantics of epistemic logic programs yield world views, which are sets of stable models. In the case of epistemic logic programs, we also say that these world views are a collection of belief sets that satisfy the rules of any given epistemic logic program. The aim of our research is to develop effective tools for the computation of such world views. As an example,

$$\begin{aligned} a &\leftarrow not\ K\ b \\ b &\leftarrow not\ K\ a \end{aligned} \tag{3a}$$

returns two different world views $\{\{a\}\}$ and $\{\{b\}\}$, each containing a single stable model. Epistemic logic program *solvers* play a pivotal role in the computation of these intricate world views. Their function extends beyond mere computation, delving into the nuanced task of unraveling and articulating the diverse sets of subjective literals that define each world view. As such, these solvers serve as indispensable tools for navigating the complexities of epistemic logic programming, providing valuable insights into the multiple perspectives and interpretations that can arise within a given

logical framework.

2 Background

This section will serve as an introduction to many topics relevant to this thesis. From a history survey about the early days of epistemic logic to the first iterations of `eclingo` and the engineering work behind its development, as well as a quick summary of the existing solvers created to solve problems in the language of epistemic specifications. While the task of comparing performance, advantages and disadvantages between these different solvers is outside of the scope of this thesis, it is important to acknowledge the work and use it as reference for the sake of developing a more robust and complete tool. On the other hand, this section will also introduce the concept of metaprogramming, which is a fundamental building block for this research project. We will explain how is currently used in the field, how it is used in `clingo`, as well as its reification capabilities. This introduction will help to lay the necessary foundations required in order to understand the scope and final objective of this work.

2.1 ASP Applications

The ASP paradigm, while relatively novel in practical applications, has already demonstrated its efficacy in diverse fields, yielding interesting and successful outcomes. One notable application extends into the realms of science and humanities, specifically within phylogenetic systematics. This scientific domain, concerned with unraveling evolutionary relationships among species based on shared traits, has leveraged ASP to enhance its analytical capabilities [9]. The utilization of ASP in phylogenetic systematics signifies a departure from traditional methods, showcasing its adaptability to varied domains beyond its initial conceptualization. On a different front, ASP has found considerable traction in industrial applications, with product configuration standing out as a prominent example [10]. In this context, ASP serves as the backbone of a system designed to generate, through rule-based mechanisms, the exhaustive space of all conceivable combinations within a given product configuration. This application not only underscores the versatility of ASP in tackling combinatorial problems but also highlights its practical relevance in addressing real-world challenges within the industrial landscape. Moreover, an iconic instance in the industrial sector is the deployment of ASP in the decision support system for the space shuttle [11], marking an early foray into leveraging ASP for complex problem-solving. In scenarios where numerous failure possibilities exist and pre-planning for every potential combination is infeasible, ASP emerges as a vital tool. The system, armed with ASP, can dynamically suggest optimal courses of action based on available information, contributing to efficient decision-making in situations of heightened complexity. ASP doesn't limit itself to a specific domain but rather adapts to various problems, showcasing its versatility and applicability across different fields. In the realm of Data Management, ASP has emerged as a valuable tool for handling querying tasks and query engines on the Web. Notably, one of the pioneering reasoning engines, SPARQL, was developed through an ASP encoding, underscoring ASP's role in advancing the capabilities of data processing and retrieval [12]. This intersection of ASP with Data Management not only demonstrates its adaptability but also contributes to the evolution of technologies

that facilitate effective data querying and manipulation. Equally significant are ASP's contributions to the field of Artificial Intelligence (AI), building upon its roots in knowledge representation and non-monotonic reasoning. ASP has played a pivotal role in early AI endeavors, tackling problems such as constraint planning, diagnosis, and agent decision-making [13]. The expansive application of ASP in AI scenarios reflects its foundational role in shaping intelligent systems and problem-solving methodologies. Looking ahead, this work anticipates making a meaningful impact in AI, particularly in areas like agent decision-making, where the incorporation of epistemic specifications is poised to enhance the efficiency of problem-solving processes.

2.2 Epistemic Specifications and Epistemic Logic Programs

The exploration of the intricate relationship between knowledge and veracity traces its roots back to the early philosophical inquiries of Aristotle and extends through the medieval era. These inquiries often centered around statements of the form '*If I know p, then p is true*'. As we progress through the historical evolution of thought, the trajectory brings us to the modern treatments of the logic of knowledge and belief. Notably, the late 1940s and the entirety of the 1950s witnessed a surge in philosophical and logical investigations into this domain. A pivotal contribution during this period was the groundbreaking work of von Wright [14], widely credited with initiating the formal study of epistemic logic as it is understood today. Building upon von Wright's foundations, Jaakko Hintikka further extended the discourse in his influential book titled *Knowledge and Belief: An Introduction to the Logic of the Two Notions* [15]. Hintikka's work introduced a novel interpretation of epistemic concepts through the lens of possible world semantics. This conceptual framework has since become a cornerstone for the study of epistemic logic from both philosophical and logical perspectives. While the initial developments in this field were somewhat detached from the concurrent advancements in theoretical computer science, a notable shift occurred in the past two decades. Over this period, epistemic logic has evolved into a comprehensive set of formal approaches, garnering increased interest from computer scientists actively contributing to the development of tools and theories. From the standpoint of philosophers, contemporary explorations in epistemic logic predominantly embrace a modal conception of knowledge. Notably, since the 1960s, Kripke's influential models [16, 17, 18] have served as fundamental building blocks for the widely adopted semantics across various modal logics. The most recent developments in epistemic logic, particularly from the 1960s onward, have been marked by a modal understanding of knowledge. This perspective interprets knowledge through the lens of informational indistinguishability between possible worlds, drawing on earlier foundational work [19]. This shift in focus underscores the interdisciplinary nature of contemporary epistemic logic, extending its reach beyond traditional realms of philosophy and logic to actively involve computer scientists in advancing both tools and theoretical frameworks.

When it comes to the computer science perspective of epistemic logic, a lot of effort has been done trying to develop a more robust semantics for epistemic logic programs. The development of Epistemic Specification, which conform the base of what is described here as the ELP language, has come a long way since Michael Gelfond first introduced it [5, 6]. Part of the following improvements are focused in the semantic subtleties of the language, particularly the ones including rules involving recursion through the defined $\mathbf{K}l$ and $\mathbf{M}l$ modal operators. As a result there are several computing

semantics with different properties. See [20] for a comprehensive survey. Here we will focus on the original semantics, usually denoted G94 [6], and the semantics K15 [21] that tried to fix the multiple world views computed due to the recursion of the operator **M** as mentioned earlier. Lastly, based upon the semantics C19 [22], developed as a consequence of the attempts to develop a semantic that satisfied the foundedness property, a new variant called C19-1 is introduced. This variant is used to implement the already mentioned G94 semantics by replacing $\&k\{a\}$ by *not not* $\&k\{a\}$ and then applying C19-1.

2.3 *eclingo*

In this thesis we aim to extend the capabilities of the epistemic solver `eclingo` [7], which is built upon the Answer set programming system `clingo` along with its multi-shot capabilities [23]. `eclingo` algorithm follows a guess and check strategy, for which, it first generates potential candidates, and then checks the obtained results with respect to its brave and cautious consequences. We plan to extend its capabilities and make use of the reification process of clingo to add new inference rules to the solving process. Originally, `eclingo` was developed in 2020 by a series of developers at the University of Corunna, Spain; and Postdam, Germany. The main idea behinds its development was to create a tool that used the ASP solver `clingo` under the hood, shared compatibility with python for a more simple use of its functionality, and was at the moment, the state-of-the-art solver for epistemic logic programs. Some of these world views might result from self-supported derivations, in that remark a lot of work has been done on the *foundedness* of epistemic logic programs [22] [24] in order to decide what is the most intuitive solution and to get rid of these self-supported world views. Taking as an example the program (2c), we observe that each answer generated by `eclingo` corresponds to a specific world view within the context of the epistemic program $\prod$. These world views are articulated through the set Z, which encapsulates the subjective literals L that are valid for a particular world view. Importantly, this set Z aligns with the answer set $AS[\prod^W]$ within the world view, as discussed in previous work [7]. The composition of the world view involves the answer set $AS[\prod^W]$ derived from $\prod_Z$. To illustrate, consider the initial solution where $\&k\{a\}$ is set to $\top$ (true), and $\&k\{b\}$ is set to $\bot$ (false). This configuration constitutes the first solution. In contrast, the second solution flips these assignments, setting $\&k\{a\}$ to $\bot$ and $\&k\{b\}$ to $\top$. The variability in these solutions reflects the dynamic nature of epistemic logic programs and the diverse world views that can emerge from them. As stated before, the `eclingo` main algorithm solves epistemic logic programs through a guess and check strategy. In the guessing phase, subjective literals L are replaced by auxiliary atoms and a regular logic program is generated. Diverse epistemic semantics have different sets of rules when generating this auxiliary atoms. The theory behind all of these semantics is outside of the scope of this thesis, but is collected and explained in the survey named 'Thirty years of epistemic specifications' [20]. However, it is good to understand that as a result of this translation we obtain a regular logic program that can now be used for guessing the truth values of subjective literals L, represented as auxiliary atoms. During the checking phase `eclingo` verifies that each and every one of the candidates generated and tested are actually a valid interpretation. In order to do that, we will proceed to check some conditions on the subjective literals L w.r.t. the answer sets of the candidate world view [20]:

1. For each subjective literal $\&k\{l\}$, literal l must be in every answer set.
2. For each subjective literal *not* $\&k\{l\}$, literal l cannot be in every answer set.
3. For each subjective literal $\&k\{\neg l\}$, literal l cannot be in any answer set.
4. For each subjective literal *not* $\&k\{\neg l\}$, literal l must be in some answer set.

After the checking, and in order to get all the answer sets of a candidate world view X, we need to expand for all the answer sets of the epistemic program $\prod_X$ that has been yield by `clingo`. The expansion step is avoided by using cautious and brave consequence reasoning, which involves computing the iterated intersection and union operations of all answer sets, respectively. By definition, let $cautious(\prod_X)$ and $brave(\prod_X)$ denote the set of atoms in the cautious and brave consequences of the epistemic program $\prod_X$, respectively. In particular, we can reduce those four conditions listed earlier to:

1. For each subjective literal $\&k\{l\}$, check $l \in cautious(\prod_X)$.
2. For each subjective literal *not* $\&k\{l\}$, check $l \notin cautious(\prod_X)$.
3. For each subjective literal $\&k\{\neg l\}$, check $l \notin brave(\prod_X)$.
4. For each subjective literal *not* $\&k\{\neg l\}$, check $l \in brave(\prod_X)$.

After this basic solving process of the original algorithm we can proceed to build the correct world view, and solve the problem whether it is satisfiable or unsatisfiable. This original version or legacy version from the point of view of this work supports both G94 and K15 semantics.

2.4 Other epistemic logic solvers

Shifting away form the theory perspective, we move our focus into more practical applications. Here, we encounter different attempts to develop solvers for ELPs. All these solvers, before the development of eclingo, work by generating an epistemic reduct framework for the ELP. All of them rely on an underlying ASP solver, whether it is DLV [25], claspD, or as `eclingo` does, `clingo` [26] that is then used to compute the answer sets of the epistemic reduct also knows as the world views. Earlier attempts at developing these solvers can be found at the summary paper: 'A survey of advances in epistemic logic program solvers' [27]. However, for the sake of simplicity regarding this work, we will focus on the first solver for which there exists a working version widely available. This example was developed for the G94 semantics was `Wviews` [28] is a solver for epistemic logic programs developed for G94 semantics, which has issues both with performance and the flexibility if its accepted language. Another of this systems is the solver called EP-ASP [29] which was a significant improve in performance for ELPs while using the K15 semantics. The main idea behind this solver was to take the epistemic reduct framework and to solve a single answer set rather than for all possible guesses like other solvers such as ELPS do [30]. This guesses are consistent. A deeper-level abstraction of such implementation can be followed in the previously mentioned paper. One of the parallels between this solver and our system `eclingo` is the use of clingo as runtime environment. Finally, we described the solver used for this thesis: `eclingo`. At the moment of its

release it outperformed the above mentioned solves by solving 21 of the 25 instances used in previous benchmarks under a second [7]. Finally, another ELP solver has been presented this year [31]. This solver rewrites epistemic logic programs as ASP program with Quantifiers. Similar to `eclingo`, this solver significantly outperform the two first solvers. Unfortunately, to the best of our knowledge no comparison between this tool and `eclingo` exists.

2.5 Metaprogramming

The term 'metaprogramming' relates to 'programming' as 'metalanguage' relates to 'language' and 'metalogic' to 'logic': programming where the data represent programs [32]. As described in the work by Bowen and Kowalski [33], the concept of metalogic programming has mostly been concerned with provability. This entails if an expression or sentence S can be derived through infinitely many derivations of different inference rules. Metaprogramming serves the dual purpose of altering the semantics of language constructs and implementing novel ones, including optimization techniques tailored for reducing search spaces. The initial concept of applying metaprogramming to solvers and extending Answer Set Programming (ASP) programs, as outlined by Kaminski [34], draws inspiration from clingo's reification feature. This concept has been further expanded upon, resulting in an original contribution to `eclingo`. In this iterative process, the original program is treated as data and fed into a meta-program or meta-encoding (refer to Section 4 for an in-depth explanation) that incorporates the latest functionality or optimization technique. Various use cases within ASP have benefited from this approach, such as optimization efforts [35], where existing statements are reinterpreted to express more intricate preferences among the generated answer sets. Additionally, metaprogramming has found application in the development of debugging tools for ASP [36], contributing to a broader acceptance of the paradigm. Furthermore, other ASP and logic program-related systems have previously leveraged metaprogramming with notable success [37, 38]. This utilization of metaprogramming not only showcases its versatility in enhancing solver capabilities but also underscores its broader impact on diverse areas within the realm of logic programming and ASP. The adaptability of metaprogramming proves instrumental in addressing varied requirements and challenges, demonstrating its efficacy across different contexts in the field.

Across the years, metaprogramming in logic has been used for different languages and interpreters with various functionalities, objectives and problem selection. Some of them were basic First-Order-Logic systems that incorporated rejection principles as inference rules [39]. Others focused primarily in the most used languages by the industry like Prolog, and were developed due to its necessities and diverse use cases like MetaProlog [40] where such extensions enabled programs to manipulate multiple theories, allowing these diverse theories to have multiple 'viewpoints' of a relation, and to create some proofs as terms in the language [41]. Another example of early work in metaprogramming comes from the field of languages based on combinatory logic. These languages lack locally bound variables, a characteristic noted by Nilsson. This attribute renders combinatory languages especially intriguing for metaprogramming, as the representation of such languages avoids the implementation issues previously discussed [42]. Nilsson delves into the metaprogramming potential of combinatory logic by introducing a metainterpreter designed for a proposed combinator logic programming language. All of this previous work focused on the development of metaprogramming as a theory and the creation

of systems such as interpreters for diverse applications. Such applications of metaprogramming in logic can be summarize into four main categories such as metaprogramming as a formalism for compilers, program transformers, debuggers and abstract interpreters that allow for developing these writing program manipulation tools. Another category is the control of procedural behavior within logic programs [43]. Thirdly, it has also been use to develop, as I mentioned before, interpreters for languages [44], whether that involved new programming languages or just new functionality. Lastly, but arguably the most important in relation to the use cases that we hope the system `eclingo` could have in the future is metaprogramming for representing partial or complete knowledge [45, 46], reasoning about it, or even reasoning about reasoning (meta-reasoning).

3 Motivation

This research endeavor originated from the aspiration to pioneer the next generation of Epistemic Logic Program (ELP) solvers. Our overarching objective was to enhance the already advanced ELP solver `eclingo` by delving into the realm of metaprogramming, positioning it as the foundational requirement for the development of our newer version. In pursuit of this goal, we dedicated our efforts to not only improving the existing state-of-the-art solver `eclingo` but also envisaging the potential for extending its functionality through the incorporation of *metaprogramming* capabilities. To elucidate, *metaprogramming*, as detailed in the preceding section and referenced by Kaminski [34], is a technique that treats other programs as data that can be manipulated by another Answer Set Programming (ASP) program. This innovative approach offers the prospect of introducing new inferences into the ELP solver by modifying an ASP program in the ASP language itself obviating the need to develop an entirely new program. The adoption of metaprogramming in our research opens up avenues for dynamic and adaptable enhancements to `eclingo`, fostering a more flexible and extensible framework. The anticipation is that this strategic integration of metaprogramming capabilities will not only facilitate the implementation of novel ideas but also yield substantial performance improvements for ELP solvers. By leveraging the malleability inherent in metaprogramming, we aspire to cultivate an environment where `eclingo` can evolve and adapt to emerging challenges, thereby solidifying its position as an innovative and high-performing solution in the realm of Epistemic Logic Programs.

4 Implementation and Original Contribution

Our research focuses on implementing metaprogramming techniques with the goal of developing new inferences that enhance efficiency compared to previous versions and other epistemic solvers. Converting programs into data poses a significant challenge, but this hurdle is effectively addressed by the underlying grounder of *clingo* known as *gringo*. *Gringo* creates a fact-based representation of the grounded logic program, resolving the issue by employing a process called reification, which involves transforming a (ground) logic program into a set of facts.

The overall implementation is structured into two distinct phases. The initial phase involves preprocessing the program through parsing and the reification step. In this stage, the logic program

is transformed into a set of facts. The subsequent phase encompasses the solver algorithmic process, which generates computed world views based on the preprocessed and reified program. This two-step approach enables a systematic and efficient application of metaprogramming techniques in our research.

4.1 Parsing the AST

The very first step in our novel implementation comes from the need of representing the epistemic operators previously introduced by eclingo (**K** and **M**) in such a way that the underlying ASP solver `clingo` is able to manipulate them as symbols in the standard language of ASP itself.. Therefore, the first thing to do is to parse the given logic program in the language of epistemic specifications and modify the Abstract Syntax Tree, also known as *AST*, to introduce these epistemic operator. The AST is a tree representation of the abstract syntactic structure of text (the source code symbols) written in a formal language. Each node of the tree denotes a construct of the ASP language occurring in the program. There are many, such as atoms (atomic formula), literals, functions, term, and much more as described in [47]. Some of these concepts can be defined such like, by term we understand a constant symbol, a variable symbol or a function symbol followed by a sequence of terms [32]. On the other hand, an atomic expression or sentence, or for the same argument a simple atom, can mean a propositional constant symbol or predicate. The literal is an atomic sentence or the negation. For instance, subjective literals in the syntax of `eclingo` are written as:

$$\begin{aligned} \&k\{a(X)\} \\ \&k\{not\ a(X)\} \\ \&k\{not\ not a(X)\} \end{aligned} \tag{4a}$$

These expressions are first converted into regular ASP atoms of the form:

$$\begin{aligned} k(a(X)) \\ k(not1(a(X))) \\ k(not2(a(X))) \end{aligned} \tag{4b}$$

During these transformations, the parsing process takes the original AST of any program, for example: &k{ a }, and rewrites the Abstract Syntax Tree as:

where example *number 1* represents the *ast.Function* representation as a node on the AST which arguments are the location, the string representation of the name of the atom: 'a', the Sequence of AST terms being the arguments of the atom predicate 'a', and a truth value representing if the atom is positive or negative. On the other hand, example *number 2* represents the *clingo.symbol.Function* implementation that constructs a function symbol. Identical to the first with the exception of excluding the location argument. Both implementations are used simultaneously during the implementation. All this process is obtained by implementing function transformers that modify the existing Abstract Syntax Tree representation of the rules and statements of the epistemic logic programs as explained above. Once, the parsing has been completed for all epistemic literals on the program, it can be treated as a *clingo* program in the ASP language. Then, the reification process

AST Rewritten

```
1.
    ast.Function(
        Location(begin=Position(), end=Position()),
        'k',
        [ast.Function(Location(begin=Position(),
                               end=Position()),
                      'a', [], 0)], 0
    )

2.
    [Function('k',
        [Function('u',
            [Function('a', [], True)],
        True)],
    True)]
```

Figure 1: AST parsing for a given ELP.

can begin.

4.2 Reification

A complete epistemic logic program consisting of a single subjective constraint such as

$$b \leftarrow \&k\{not\ a\}. \tag{5a}$$

will be expressed as

$$\begin{aligned} \{\, k(not1(u(a)))\,\} &\leftarrow not1(u(a)). \\ not1(u(a)) &\leftarrow not\ u(a). \\ u(b) &\leftarrow k(not1(u(a))). \end{aligned} \tag{5b}$$

after being parsed and converted to a regular clingo program. Later on, the reification process takes place during the grounder step. Once the program has been parsed, it is grounded while the reified-grounded facts are registered through an observer that will gather the symbols of the reification. This observer is the *Reifier* from the `clingox` library. After that, the facts are parsed into a string to be used in the solver process. Taking as an example the program introduced in (4a) and reifiying it

$$b \leftarrow \&k\{not\ a\}.$$

we obtain the following group of facts

$atom\_tuple(0)$.	$atom\_tuple(0,\ 1)$.	$literal\_tuple(0)$.
$rule(disjunction(0),\ normal(0))$.	$atom\_tuple(1)$.	
$atom\_tuple(1,\ 2)$.	$rule(choice(1),\ normal(0))$.	$atom_tuple(2)$.
$atom\_tuple(2,\ 3)$.	$literal\_tuple(1)$.	$literal\_tuple(1,\ 2)$.
$rule(disjunction(2),\ normal(1))$.	$output(k(not1(u(a))), 1)$.	
$literal\_tuple(2)$.	$literal\_tuple(2,\ 3)$.	
$output(u(b), 2)$.	$output(not1(u(a)), 0)$.	

These facts, and their meaning [34], are representing the format that is obtained from a non-epistemic logic program using the command `--output=reify` while using clingo. On this new implementation the reification step is processed by default substituting the previous original implementation. Therefore, *eclingo* will process any epistemic logic program and convert it into a reified representation.

4.3 Solver Algorithm

This section aims to present an overview of the primary algorithm at the core of eclingo, referred to as **Algorithm 1**. The **Algorithm 1** serves as a comprehensive abstraction of the original implementation, delineating the step-by-step procedure that guides the entire computational process. Originally conceived during the development of the initial version of eclingo, the fundamental essence of **Algorithm 1** has been retained. Therefore, from a bigger perspective the procedure looks the same. However, the original work contributed by this thesis introduces several modifications across the three primary components or steps that made up the algorithm. These components are the Candidate Generator, the Candidate Tester and the World View Builder.

On top of that, in its current form, the updated algorithm takes as input the original epistemic logic program for which a solution is sought. Notably, the input now adopts the structure of a set of reified facts, as generated from the preceding reification step elucidated earlier. This adjustment marks a significant transformation in the algorithm's data handling approach. Following this, the main eclingo algorithm is executed, leveraging the groundwork laid by the reified facts to deduce solutions.

Algorithm 1: Eclingo Solver main algorithm

```
Data: Reified Facts
Result: ELP World Views
Initialize World View Builder, Tester and Generator.;
for candidate in GenerateCandidate(candidate) do
    candidate_bool ← TestCandidate(candidate);
    if candidate_bool is True then
        yield BuildWorldView(candidate);
    end
end
```

Solving an epistemic logic program involves the correct execution of three main sub-modules

Generator Base Meta-Program

```
conjunction(B) :- literal_tuple(B),
                  hold(L) : literal_tuple(B, L), L > 0;
                  not hold(L) : literal_tuple(B, -L), L > 0.

body(normal(B)) :- rule(_, normal(B)), conjunction (B).
body(sum(B, G)) :- rule(_, sum(B,G)),
                #sum {
                    W,L : hold(L),
                    weighted_literal_tuple(B, L,W), L>0;
                    W,L : not hold(L),
                    weighted_literal_tuple(B, -L,W), L>0
                } >= G.

fact(SA) :- output(SA, LT),
            #count {
                L : literal_tuple(LT, L)
            } = 0.

positive_candidate(k(A)) :- fact(k(A)).
positive_candidate(k(A)) :- output(k(A), B), conjunction(B).
negative_candidate(k(A)) :- output(k(A), B), not conjunction(B).

atom_map(SA, A) :- output(SA,LT),
                #count {
                    LL : literal_tuple(LT, LL)
                } = 1,
                literal_tuple(LT, A).
strong_negation_complement(A, B) :- atom_map(u(SA), A),
                                    atom_map(u(-SA), B).
:- hold(A), hold(B), strong_negation_complement(A, B).

#show positive_candidate/1.
#show negative_candidate/1.
```

Figure 2: Metaprogramming template example used in the Generator component.

or components. Each of these components has been modified and updated to support the reification implementation. To begin with, we will introduce the candidate generation process and its optimization meta-encodings.

4.4 Generator

The generator is in charge of receiving a copy of the reified set of facts from the incoming ELP, and then injecting a base metaprogramming ASP template. In the process, certain other meta-programs are added to optimize the performance including a fact optimization program that propagates the facts into epistemic facts, and is added to the control module. These templates can be seen as part of the **Appendix A** at the end of the document.

The last previous code snippet is in charge of propagating facts into epistemic facts, needed to boost the performance of the generation of the correct world view candidates. Lately, based in the model generated by the grounding and solving of the metaprogramming templates we calculate the

candidates. The totality of the *Generator* procedure can be followed in **Algorithm 2**.

Algorithm 2: Generate Candidates

```
Data: Reified Program
Result: Candidates
Initialize Generator;
begin
    InitControl(reifiedProgram);
    controlHandle ← Solve(metaProgrammingTemplates);
    for model in controlHandle do
        candidate ← CreateCandidate(model) if candidate is Positive then
            candidate.pos ← candidate;
        else
            candidate.neg ← candidate;
        end
        return Candidate[candidate.pos,candidate.neg]
    end
end
```

4.5 Tester

The *Tester* is responsible for checking that the generated candidates by the *Generator* are actually part of the calculated model. Before testing the assumptions, it also injects a modified metaprogramming (Tester Base Meta-Program) and grounds it, that will create the model which the candidates will be tested upon. In this model there will also be facts related with the *#show* directive. This directive is part of clingo, and is used to only return the world views that contain the atoms referenced by the directive. For example,

$$a \leftarrow not\, a$$
$$b \leftarrow not\, K\, a$$
$$c \leftarrow not\, K\, b$$

when using "*#show b/0.*" will yield a world view such as $\&k\{b\}$ when the complete world view should also yield $\&k\{a\}$. Also, including a show statement for the predicate atom a will also include the mentioned epistemic literal of the same atom. The full Test Candidates algorithm is shown in **Algorithm 3**. On top of the meta-program used, the tester includes a couple of preprocessing techniques to approximate the stable models of the program. In that aspect, it will return none stable models when the problems is determined unsatisfiable. On the other hand, for heavy-computational problems, it will return an approximation of the stable models of the program in a pair of sequence of symbols where the atoms contained in the first sequence are true, and false (in all stable models) otherwise. This approximation technique runs in polynomial time. During the testing phase, the reasoning is being done by the cautious consequences of the program.

Tester Base Meta-Program

```
conjunction(B) :- literal_tuple(B),
                  hold(L) : literal_tuple(B,  L),
                            L > 0;
                  not hold(L) : literal_tuple(B, -L),
                                L > 0.
body(normal(B)) :- rule(_, normal(B)), conjunction (B).
body(sum(B, G)) :- rule(_, sum(B,G)),
                   #sum {
                        W,L : hold(L),
                        weighted_literal_tuple(B, L,W), L>0;
                        W,L : not hold(L),
                        weighted_literal_tuple(B, -L,W), L>0
                    } >= G.
hold(A) : atom_tuple(H,A) :- rule(disjunction(H), B), body(B).
{hold(A) : atom_tuple(H,A)} :- rule(choice(H), B), body(B).

atom_map(SA, A) :- output(SA,LT),
                #count{
                    LL : literal_tuple(LT, LL)
                } = 1, literal_tuple(LT, A).

strong_negatation_complement(A, B) :- atom_map(u(SA), A),
                                      atom_map(u(-SA), B).
:- hold(A), hold(B), strong_negatation_complement(A, B).

symbolic_atom(SA) :- atom_map(SA, _).

symbolic_epistemic_atom(k(A)) :- symbolic_atom(k(A)).
epistemic_atom_map(KSA, KA) :- atom_map(KSA, KA),
                               symbolic_epistemic_atom(KSA).
epistemic_atom_int(KA) :- epistemic_atom_map(_, KA).

symbolic_objective_atom(OSA) :- symbolic_atom(OSA),
                                not symbolic_epistemic_atom(OSA).

has_epistemic_atom(A) :- symbolic_epistemic_atom(k(A)).

fact(SA) :- output(SA, LT), #count {L : literal_tuple(LT, L)} = 0.

hold_symbolic_atom(SA) :- atom_map(SA, A), hold(A).
hold_symbolic_atom(SA) :- fact(SA).

u(SA)    :- fact(u(SA)).
u(SA)    :- output(u(SA), B), conjunction(B).
not1(SA) :- output(not1(SA), B), conjunction(B).
not2(SA) :- output(not2(SA), B), conjunction(B).

{k(A)} :- output(k(A), _).

hold(L) :- k(A), output(k(A), B), literal_tuple(B, L).
:- hold(L) , not k(A), output(k(A), B), literal_tuple(B, L).

preprocessing_hold(KA) :- epistemic_atom_map(k(SA), KA),
                          hold_symbolic_atom(SA).
```

Figure 3: Metaprogramming template example used in the Tester component.

Algorithm 3: Test Candidates

```
Data: Reified Program, Generated Candidate
Result: Bool
InitTester(reified_program);
begin
    for model in controlHandle do
        for atom in candidate.pos do
            if not model.contains(atom) then
                return False
            end
        end
        for atom in candidate.pos do
            if model.contains(atom) then
                return False
            end
        end
    end
    return True
end
```

4.6 World View Builder

The last part of the solving process is to create and process the corresponding world views based on the stable models of the program. In this aspect, the **World View Builder** will also perform an injection of an meta-program encoding similar to the Tester Base Meta-Program on top of the original reified program. After the generation of the model, it will construct the world view from the generated and tested candidates, therefore being the third and final step.

Along this step, the epistemic operators **K** and **M** will be generated based on the candidates. If the candidates are positive, then it will generate the **K** symbol which implies that the corresponding atom is true in every stable model. On the other hand, when the candidate is negative, it will build the **M** symbol for the given candidate atom, which implies that it is only true in some stable models.

As part of the process, after the model is created and the candidates have been assumed there is a check procedure for given show statements in the program, which is indeed responsible for yielding the particular predicates as explained in section 4.5. This step looks for the intersection of *show_statement* facts within the model and the candidates. Once, there is a match, it means that the world view to be yielded should include and show the correct atom corresponding to the arguments of the *show_statement*. The full description of the process can be observed on **Algorithm 4.**

5 Benchmarking

Upon successful implementation of the enhanced version of **eclingo**, incorporating the improvements detailed earlier and subjecting it to rigorous testing, the subsequent step involves benchmarking and comparing its performance with the older version. The benchmarking process entails measuring the time, in seconds, required by each version to solve various instances. Two distinct batteries of tests, well-established in the literature, have been employed for this purpose: *'The bomb problems'* and *'The Yale shooting problems'*. While these benchmarks offer comprehensive insights into the

Algorithm 4: World View Builder

```
Data: Reified Program,
Result: World Views
InitWorldViewBuilder(reifiedProgram);
WorldViewFromCandidate(candidate);
begin
    for literal in candidate do
        show_ep_lit ← EpistemicShowStm(model);
        begin
            if model.contains(show_statement_atom) then
                new_candidates ← show_statement_atom;
                return new_candidates
            end
        end
        if candidate is Positive then
            epistemic_literals ← GenerateSymbolK(candidate.pos);
        else
            epistemic_literals ← GenerateSymbolM(candidate.neg);
        end
    end
    return show_ep_lit OR epistemic_literals
end
```

solver's capabilities within the scope of epistemic logic programming, it is noteworthy that other benchmarks discussed in the literature, such as the reversibility problem, were deemed beyond the scope of this project but remain viable for future experimental runs. The reversibility problem, within the Automated Planning sub-field, involves planning with actions having non-deterministic effects, contributing to dead-end detection and recovery from undesirable action effects [48]. Another discarded problem, the eligibility problem, gauges whether a person qualifies for an interview based on provided knowledge, albeit its relatively easy computational nature makes it less suitable for performance comparison between the newer and older eclingo versions. In executing the benchmarking process, a Linux machine equipped with an E5520 CPU boasting 24 cores at 2.27GHz and 24 gigabytes of memory served as the computational platform. To enhance efficiency, each run was parallelized into four sequential threads with a timeout set at 600 seconds. Furthermore, every instance underwent computation twice, and the final result, expressed in seconds, is the average of both runs. This approach minimizes the impact of noise, attributing any time disparities to insignificant variations. Instances that failed to produce a world view within the allocated timeout are denoted with a - symbol. Notably, results presented in bold signify the faster performance when comparing the implementation in this thesis with the older version (non-reification). In summary, this benchmarking initiative stands as a crucial evaluation of the enhanced eclingo, shedding light on its efficiency and improvements over the previous version across various problem instances. The careful selection of benchmarks ensures a comprehensive assessment of its performance within the domain of epistemic logic programming.

	Non-Reification	*Reification*
bomb_fail_10	-	**3.92411**
bomb_fail_20	-	**12.7746**
bomb_fail_30	-	**29.7195**
bomb_fail_40	-	**52.1243**
bomb_fail_50	**11.4801**	84.196
bomb_fail_60	178.683	**140.053**
bomb_fail_70	-	**182.522**
bomb_fail_80	-	**240.571**
bomb_fail_90	-	**322.669**
bomb_fail_100	-	**409.645**
bomb_fail_110	-	**498.165**
bomb_fail_120	-	-
bomb_fail_130	-	**515.184**
bomb_fail_140	-	**558.948**
bomb_fail_150	-	**379.334**

	Non-Reification	*Reification*
bt_bomb_10	-	**3.42815**
bt_bomb_20	-	**14.8314**
bt_bomb_30	-	**32.5565**
bt_bomb_40	-	**51.7028**
bt_bomb_50	-	**77.5847**
bt_bomb_60	-	**113.813**
bt_bomb_70	-	**163.369**
bt_bomb_80	-	**216.868**
bt_bomb_90	-	**301.304**
bt_bomb_100	-	**375.712**
bt_bomb_110	-	**489.165**
bt_bomb_120	-	-
bt_bomb_130	-	**481.867**
bt_bomb_140	-	**563.376**
bt_bomb_150	-	**579.753**

Table 1: Bomb Problem (bt_base.lp base encoding). Left: bomb_fail encoding. Right: bt encoding. Time in seconds.

5.1 Bomb Problems

The Bomb problems, also known as the "bomb in the toilet" (BT) problem, were initially conceptualized by Reichgelt and defined for the first time in his work [49]. Reichgelt also introduced variations of this experiment, denoted as BT(p), B1TC(p), B2TC(p), B3TC(p), and BTUC(p). In the context of these problems, the variable 'p' signifies the number of packages in a given scenario. The fundamental premise of the bomb in the toilet problem posits that, to be considered 'safe,' one must submerge all the packets into one of the available toilets, as exemplified by BT and B1TC(1). Notably, the act of dunking a packet introduces an element of uncertainty, as it may or may not clog the toilet (C and UC). Moreover, the protocol mandates flushing the clogged toilet before dunking another packet into it [29]. The Epistemic Logic Programming (ELP) encoding of these problems stems from the world-state encoding elucidated in the early work by Eiter [50], although the detailed explanation of this encoding is beyond the scope of the current thesis. Concerning the benchmark, the bomb problems will assume new names while preserving the inherent challenge articulated in the prior works. The problems are categorized into two sets of tests. The first set, termed the base tests, includes btc_bomb, btuc_bomb, bomb_fail, and bt_bomb. These tests are executed using a foundational encoding labeled bt_base.lp. For the purpose of reproducibility, each instance can be run independently using the command: *time eclingo bt_base.lp base_test bomb_instance*. This benchmarking approach allows for systematic evaluation and comparison of the various instances, facilitating a comprehensive understanding of the performance and capabilities of the solver in tackling different bomb problems.

Both table 1 and table 2 represent this first battery of tests within the bomb problems. Both of them are tested on 15 different instances.

	Non-Reification	*Reification*
btc_bomb_10	170.656	**7.09668**
btc_bomb_20	-	**45.4334**
btc_bomb_30	-	**60.279**
btc_bomb_40	-	**235.441**
btc_bomb_50	-	**519.983**
btc_bomb_60	-	**371.753**
btc_bomb_70	-	-
btc_bomb_80	-	-
btc_bomb_90	-	-
btc_bomb_100	-	-
btc_bomb_110	-	-
btc_bomb_120	-	-
btc_bomb_130	-	-
btc_bomb_140	-	-
btc_bomb_150	-	-

	Non-Reification	*Reification*
btuc_bomb_10	-	**2.63617**
btuc_bomb_20	-	**8.87296**
btuc_bomb_30	-	**24.3665**
btuc_bomb_40	-	**216.936**
btuc_bomb_50	-	-
btuc_bomb_60	-	-
btuc_bomb_70	-	**133.711**
btuc_bomb_80	-	-
btuc_bomb_90	-	-
btuc_bomb_100	-	-
btuc_bomb_110	-	-
btuc_bomb_120	-	-
btuc_bomb_130	-	-
btuc_bomb_140	-	-
btuc_bomb_150	-	-

Table 2: Bomb Problem (bt_base.lp base encoding). Left: btc encoding. Right: btuc encoding. Time in seconds.

	Non-Reification	*Reification*		*Non-Reification*	*Reification*		Non-Reification	Reification
bmtc_bomb_10_01	**0.986218**	2.03569	bmtc_bomb_60_01	-	-	bmtc_bomb_110_01	-	-
bmtc_bomb_10_02	107.459	**3.1768**	bmtc_bomb_60_02	-	**214.03**	bmtc_bomb_110_02	-	**553.692**
bmtc_bomb_10_03	-	**3.94052**	bmtc_bomb_60_03	-	**341.315**	bmtc_bomb_110_03	-	**544.761**
bmtc_bomb_10_04	11.4287	**3.95527**	bmtc_bomb_60_04	-	**265.045**	bmtc_bomb_110_04	-	-
bmtc_bomb_20_01	-	**6.41091**	bmtc_bomb_70_01	-	-	bmtc_bomb_120_01	-	-
bmtc_bomb_20_02	-	**9.68625**	bmtc_bomb_70_02	-	**341.539**	bmtc_bomb_120_02	-	**435.733**
bmtc_bomb_20_03	-	**11.3516**	bmtc_bomb_70_03	-	**271.589**	bmtc_bomb_120_03	-	-
bmtc_bomb_20_04	-	**15.3137**	bmtc_bomb_70_04	-	**376.48**	bmtc_bomb_120_04	-	-
bmtc_bomb_30_01	-	**240.014**	bmtc_bomb_80_01	-	-	bmtc_bomb_130_01	-	-
bmtc_bomb_30_02	-	**31.1682**	bmtc_bomb_80_02	-	**349.006**	bmtc_bomb_130_02	-	**529.05**
bmtc_bomb_30_03	-	**46.8721**	bmtc_bomb_80_03	-	**240.572**	bmtc_bomb_130_03	-	-
bmtc_bomb_30_04	-	**46.9794**	bmtc_bomb_80_04	-	**428.896**	bmtc_bomb_130_04	-	**531.762**
bmtc_bomb_40_01	-	-	bmtc_bomb_90_01	-	-	bmtc_bomb_140_01	-	-
bmtc_bomb_40_02	-	**102.99**	bmtc_bomb_90_02	-	**399.81**	bmtc_bomb_140_02	-	-
bmtc_bomb_40_03	-	**104.079**	bmtc_bomb_90_03	-	**467.183**	bmtc_bomb_140_03	-	-
bmtc_bomb_40_04	-	**59.8161**	bmtc_bomb_90_04	-	**465.097**	bmtc_bomb_140_04	-	-
bmtc_bomb_50_01	-	-	bmtc_bomb_100_01	-	-	bmtc_bomb_150_01	-	**434.681**
bmtc_bomb_50_02	-	**209.556**	bmtc_bomb_100_02	-	**362.671**	bmtc_bomb_150_02	-	-
bmtc_bomb_50_03	-	**200.486**	bmtc_bomb_100_03	-	-	bmtc_bomb_150_03	-	**563.649**
bmtc_bomb_50_04	-	**233.942**	bmtc_bomb_100_04	-	-	bmtc_bomb_150_04	-	**353.643**

Table 3: Bomb Problem - Many instances(bt_base.lp base encoding). bmtc encoding. Time in seconds.

	Non-Reification	*Reification*		*Non-Reification*	*Reification*		Non-Reification	Reification
bmtuc_bomb_10_01	72.9668	**2.02781**	bmtuc_bomb_60_01	-	-	bmtuc_bomb_110_01	-	-
bmtuc_bomb_10_02	-	**2.75428**	bmtuc_bomb_60_02	-	-	bmtuc_bomb_110_02	-	**402.408**
bmtuc_bomb_10_03	**1.02062**	4.25005	bmtuc_bomb_60_03	-	**126.131**	bmtuc_bomb_110_03	-	**596.933**
bmtuc_bomb_10_04	**1.82674**	4.38011	bmtuc_bomb_60_04	-	**166.723**	bmtuc_bomb_110_04	-	-
bmtuc_bomb_20_01	-	**8.59636**	bmtuc_bomb_70_01	-	-	bmtuc_bomb_120_01	-	-
bmtuc_bomb_20_02	-	**9.83558**	bmtuc_bomb_70_02	-	-	bmtuc_bomb_120_02	-	**467.185**
bmtuc_bomb_20_03	-	**15.3151**	bmtuc_bomb_70_03	-	**182.575**	bmtuc_bomb_120_03	-	-
bmtuc_bomb_20_04	-	**18.778**	bmtuc_bomb_70_04	-	**249.598**	bmtuc_bomb_120_04	-	**480.239**
bmtuc_bomb_30_01	-	**72.5791**	bmtuc_bomb_80_01	-	-	bmtuc_bomb_130_01	-	-
bmtuc_bomb_30_02	**44.7558**	127.822	bmtuc_bomb_80_02	-	-	bmtuc_bomb_130_02	-	**588.83**
bmtuc_bomb_30_03	-	**143.98**	bmtuc_bomb_80_03	-	**273.893**	bmtuc_bomb_130_03	-	**473.116**
bmtuc_bomb_30_04	-	**50.4895**	bmtuc_bomb_80_04	-	**381.63**	bmtuc_bomb_130_04	-	**560.761**
bmtuc_bomb_40_01	-	**444.675**	bmtuc_bomb_90_01	-	-	bmtuc_bomb_140_01	-	-
bmtuc_bomb_40_02	-	-	bmtuc_bomb_90_02	-	**230.086**	bmtuc_bomb_140_02	-	-
bmtuc_bomb_40_03	-	**347.982**	bmtuc_bomb_90_03	-	**353.432**	bmtuc_bomb_140_03	-	**522.21**
bmtuc_bomb_40_04	-	**195.467**	bmtuc_bomb_90_04	-	**504.363**	bmtuc_bomb_140_04	-	**334.492**
bmtuc_bomb_50_01	-	-	bmtuc_bomb_100_01	-	-	bmtuc_bomb_150_01	-	**-**
bmtuc_bomb_50_02	-	-	bmtuc_bomb_100_02	-	**309.018**	bmtuc_bomb_150_02	-	**482.164**
bmtuc_bomb_50_03	-	-	bmtuc_bomb_100_03	-	**436.341**	bmtuc_bomb_150_03	-	**578.102**
bmtuc_bomb_50_04	-	**125.095**	bmtuc_bomb_100_04	-	-	bmtuc_bomb_150_04	-	**298.081**

Table 4: Bomb Problem - Many instances(bt_base.lp base encoding). bmtuc encoding. Time in seconds.

The second battery of tests for bomb problems are referred as 'many', and they include bmtc_bomb and bmtuc_bomb encodings. Both of them are tested on 60 different instances where the facts input_length(X) and input_toilet(Y) changes and is indicated by the two integers of the instance name where X is wrote first, and Y second. This second subset of tests is indicated by Tables 3 and 4. (Horizontally represented for a clearer understanding of the results)

5.2 Yale Shooting Problems

The Yale shooting problem poses a quandary in formal situational logic, challenging early logical solutions to the frame problem. It was coined by its originators, Steve Hanks and Drew McDermott [51], during their tenure at Yale University. The original described scenario involves Fred (later revealed to be a turkey) initially alive and a gun initially unloaded. The sequence of loading the gun, a pause, and subsequent firing is anticipated to result in Fred's demise. However, when inertia is represented in logic by minimizing changes in the situation, a unique proof of Fred's death after the loading, waiting, and shooting phases becomes elusive. This presented a problem where one logical solution affirms Fred's demise, while another equally valid solution postulates a mysterious unloading of the gun, allowing Fred to survive. Technically, this scenario is delineated by two fluents, which are conditions subject to truth value changes over time: 'alive' and 'loaded.' Initially, the first condition holds true, and the second is false. Subsequently, the gun is loaded, time elapses, and the gun is discharged. Such intricacies are formalized in logic by considering four time points (0, 1, 2, 3) and transforming each fluent, like 'alive,' into a time-dependent predicate denoted as 'alive(t).'

	Non-Reification	*Reification*
yale01	**0.458275**	0.638951
yale02	**0.57397**	0.588027
yale03	**0.480068**	0.879887
yale04	**0.454746**	0.864776
yale05	**0.548475**	1.66872
yale07	**1.88912**	2.09788
yale08	**0.970348**	2.62737

Table 5: Yale Shooting Problem (Yale.lp base encoding). Time in seconds.

Nowadays, this problem is widely used in the literature, not only as benchmark but also as explanatory example. Regarding Epistemic Logic, the adaptation of this test to conform with epistemic specifications is outside of the scope of this thesis, and was imported from the original eclingo paper. In terms of benchamrking, the Yale shooting problems consist of a single set of tests and two different encodings per instance. Table 6 uses the base encoding, while table 7 does use the variation named yale-parameter, which is expected to be slightly computationally harder than the original version. For reproducing purposes, every instance can be run independently following the

command: *time eclingo yale_test yale_instance*

	Non-Reification	*Reification*
yale01	0.48885	**0.427558**
yale02	**0.476465**	0.655435
yale03	**0.416495**	0.416591
yale04	**0.387305**	0.61013
yale05	0.515332	**0.466441**
yale07	**0.4639**	0.648343
yale08	**0.496955**	0.635298

Table 6: Yale Shooting Problem (Yale-parameter.lp base encoding). Time in seconds.

6 Results Discussion

Based on the results provided by the benchmark tool, we draw some interesting and inspiring conclusions that will be discussed in this section. Firstly, concerning the yale shooting problems, we observe an ambiguous but informative result. The base yale.lp encoding reveals that the older version (also known as non-reification on Figure 4) is still faster than the new version. The most plausible explanation for this behavior, as expected, is likely due to the grounding time taken by the reification version. When reviewing the total time taken for some of these instances, we observe that the time it takes to solve any of these problems is relatively inferior to the time it takes to compute the grounding. For benchmarking purposes, our focus is solely on the total final time it takes to either yield the answer or be determined unsatisfiable by timeout. However, this approach highlights one of the issues of the new implementation.

On the other hand, we also observe that for some of the tests based on the yale-parameter.lp encoding, the reification version is faster, as seen in particular instances like yale01 and yale05. It is necessary to mention that all yale shooting problem instances are solved in under 3 seconds, noting that they are not particularly computationally hard. While some results might initially suggest that the older version performs better, this is not the case. To truly understand these results, we need to examine the general trend for several of these batteries of tests, especially as they become progressively harder.

Looking into the other test used, the bomb problems better reflect the improvement over the older version. For most bomb_fail problems, we see that the reification version improves considerably, yielding a solution in 14 out of 15 instances and being faster than the non-reification version in 13 out of 15. Additionally, only 5 out of 15 instances are solved when using the older version. For the bt_bomb tests, none of the instances for non-reification can be solved in less than 600 seconds, while almost all but one do yield when using the new version.

Moreover, for btc_bomb and btuc_bomb, all instances that are solved under the timeout constraint

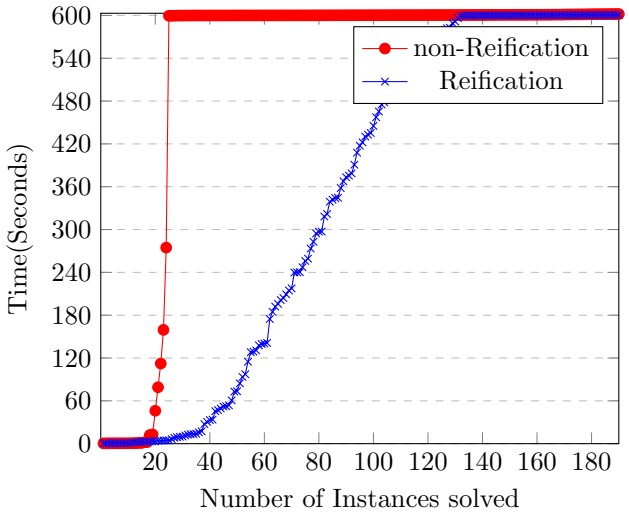

Figure 4: Version Comparison - Running Time instance analysis

are faster in the implemented version in this thesis. The same trend is observed for the 'many' set of tests, where only 1 and 3 instances of bmtc and bmtuc, respectively, were faster in the older version. This indicates that for harder problems like these, the newer version starts to show a higher success percentage and overall better performance.

In order to represent this results we will be using a well-known type of plot in the Satisfiability and SMT community also known as 'cactus' or 'survival' plots [52]. The methodology to recreate these plots is the following, for each method separately:

(a) Solve each problem p_i for time t_i (up to some threshold T).

(b) Sort the times t_i .. t_n for all n instances into increasing order of complexity.

(c) Plot the points $(t_1, 1)$, $(t_1 + t_2, 2)$, etc.

In the illustrated survival plot, the instances from both combined problems are meticulously arranged in ascending order of difficulty to solve along the x-axis, while the corresponding time taken to solve each instance is plotted along the y-axis. The red markers denote the results obtained with the older version of **eclingo**, whereas the blue markers signify the outcomes with the newly implemented version. This visual representation offers a comprehensive overview of how the solver performs across a spectrum of instances, revealing potential trends and disparities between the two versions.

The choice of a 600-second timeout for all tests is grounded in two essential considerations. Firstly, the nature of these problems, which can exhibit exponential growth in complexity. Instances have the potential to become progressively more challenging until they reach an unsatisfiable status within a human time frame, often due to grounding explosions. Consequently, a 600-second timeout provides a robust assessment of the solver's capabilities while still offering a practical representation of the challenges that real-life problems may pose.

The second rationale for the chosen timeout is specific to the field of logic programming. Comparable timeouts, typically not exceeding 1000 seconds, have been employed in similar experiments

[29]. It's noteworthy that diverse experiments, such as those outlined in the original **eclingo** paper [7], may use shorter timeouts, like 120 seconds. This exemplifies the adaptability of these experiments, with no fixed standard but rather an approach tailored to the unique characteristics of each problem. Additionally, it's crucial to highlight that each data point represented in the plot is derived from the average value (in seconds) of two independent runs of the same instance under identical conditions. This meticulous averaging approach is designed to mitigate the impact of noise, ensuring a more accurate reflection of the solver's performance across instances. In this survival plot of results we clearly observe that the version using reification and metaprogramming techniques is a better choice when the instances become harder and harder, being the only one to yield results after the 24^{th}-easiest instance which does also roughly correspond with the 3 second overhead previously mentioned. It clearly shows that for easy, non-computationally hard problems either system is good enough as the time to solve it is not a constraint. However, once the time to solve the instance starts to increase, then we see that after those 3 seconds is when one system overcomes the other as the best option.

This results express the improvement made to the solver, and the efficiency to which it computes. The slope given for the reification version grows steadily up until around the 130^{th} instance which implies that even for harder problems it still performs and yields a result within a considerably time frame. This, once again, shows the huge gap in performance from the previous version to this new one.

7 Conclusion

In conclusion, the benchmark results reveal intriguing insights into the performance of the newly implemented version compared to the older version of **eclingo**. The analysis focused on two problem sets—yale shooting problems and bomb problems. Concerning the yale shooting problems, it was observed that the older version outperformed the new version in some instances due to the grounding time taken by the reification version. However, for yale-parameter.lp encoding, the reification version showed improvement in specific instances, emphasizing the need to analyze trends across various tests.

The bomb problems provided a clearer picture of the new version's enhancement, particularly in bomb_fail instances, where the reification version outperformed the non-reification version in both speed and solution yield. Similarly, btc_bomb and btuc_bomb instances showed improved performance in the implemented version for harder problems, indicating a higher success rate and overall better efficiency.

The survival plot visualized these trends, showcasing that the reification version with metaprogramming techniques excels, especially as problem instances become more challenging. The choice of a 600-second timeout for tests was justified by the potential exponential growth in problem complexity and aligned with conventions in logic programming experiments. The plot also highlighted the adaptability of experimental timeouts across different studies.

In summary, the benchmark results affirm the significant improvement in solver efficiency with the introduction of reification and metaprogramming techniques. The performance gap between the old and new versions becomes especially evident as computational challenges intensify, emphasizing

the new version's superiority in handling complex problem instances within a reasonable time frame.

8 Impact and Future Work

The impact of this work extends beyond Answer Set Programming solvers to encompass other logic programming approaches and their respective solvers or systems. The integration of metaprogramming techniques is not confined to our implementation alone; it presents an alternative perspective that might find its way into future iterations of diverse systems. Moreover, in the realm of epistemic logic programs, we anticipate the emergence of new competitors to our solver, `eclingo`, leveraging these optimizations to advance their objectives. Considering the observed performance boost, it is conceivable that problems previously deemed unsolvable by current methods in the domain of ELP could now be addressed and ultimately resolved. One notable example involves computational problems like planning, revealing how simple epistemic planning tasks can be polynomially translated into classical planning tasks, with their complexity falling into the *PSpace*-complete domain [53]. While some instances of these problems may be deemed unsatisfiable within human time constraints, a sophisticated solver like `eclingo` could assist by reducing the number of such instances. It could serve as another valuable tool in the ecosystem of these problems, although substantial work remains to be done.

On this thesis we have provided the bones and structure of a better, more efficient and faster implementation of the epistemic logic solver known as **eclingo**. Clearly, we observe that some of the future work has been naturally inherited from this improvement. Some of this issues are purely on the engineering side of it, while others require from new experiments and results. The current algorithm (developed for the newer version) performs grounding multiple times along the full computation process. This is not ideal and creates a noisy overhead for most of the tests and benchmark results. While this situation does not provide an answer that lives to be 100% efficient, it is still better than the previous version. The first of the future steps will be to reduce the number of times grounding is executed, or to remove all of the unnecessary instances of it. In order to do this, we will need to inject the metaprogramming encodings directly to the back-end of the solver. The current implementation receives the encodings in string format, and as explained previously parses, translates and injects it into the main components at work during the algorithm. Therefore, by adding the rules of the meta encoding as part of the system without having to treat them, initially, as another program we could aim to speed up this process by a large factor of time. However, this engineering and optimizing step of the *tester* and *world view builder* components of the algorithm to execute grounding less times and to compute in faster times was originally left out of the development due to the cost versus reward payoff. Most instances that were used for testing will not observe a drastic improvement in performance, particularly for those problems that are easy to solve. However, one of the possible advantages and the reason why this improvement should be taken care of in the future is the capability of bring some of the unsolvable (under the given timeout) instances to solvable times. In the context of future iterations, it is imperative to consider the development and execution of a more extensive set of problem instances for benchmarking purposes. While the current thesis focuses on the use and comparison of two versions of the same solver on well-established problems, ensuring a robust and reliable system necessitates the demonstration of its capabilities across a more

diverse range of tests. This expansion in testing scenarios will contribute to a more comprehensive understanding of the solver's efficacy.

Furthermore, as part of our short-term objectives, it is crucial to take the results obtained in this thesis and subject them to a comparative analysis against other solvers for epistemic logic programs that were previously mentioned. Notably, `eclingo` stood out as arguably the best solver for Epistemic Logic Programs (ELP) even before the development of this new version, as evidenced in the original paper. However, our responsibility now is to substantiate this claim with more robust evidence. This involves employing the newly proposed approach and rigorously testing it against a variety of other existing solvers, thereby fortifying the assertion of `eclingo`'s superiority in the realm of ELP solvers.

References

[1] Michael Gelfond and Yulia Kahl. *Knowledge representation, reasoning, and the design of intelligent agents: The answer-set programming approach*. Cambridge University Press, 2014.

[2] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, volume 88, pages 1070–1080. Cambridge, MA, 1988.

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

[4] Wiktor Marek and Mirosław Truszczyński. Autoepistemic logic. *Journal of the ACM (JACM)*, 38(3):587–618, 1991.

[5] Michael Gelfond. Strong introspection. In *Proceedings of the ninth National conference on Artificial intelligence-Volume 1*, pages 386–391, 1991.

[6] Michael Gelfond. Logic programming and reasoning with incomplete information. *Annals of mathematics and artificial intelligence*, 12:89–116, 1994.

[7] Pedro Cabalar, Jorge Fandinno, Javier Garea, Javier Romero, and Torsten Schaub. eclingo: A solver for epistemic logic programs. *Theory and Practice of Logic Programming*, 20(6):834–847, 2020.

[8] Miroslaw Truszczynski. Revisiting epistemic specifications. *Logic programming, knowledge representation, and nonmonotonic reasoning*, 6565:315–333, 2011.

[9] Esra Erdem. Applications of answer set programming in phylogenetic systematics. In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning: Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday*, pages 415–431. Springer, 2011.

[10] Timo Soininen and Ilkka Niemelä. Developing a declarative rule language for applications in product configuration. In *Practical Aspects of Declarative Languages: First International Workshop, PADL'99 San Antonio, Texas, USA, January 18–19, 1999 Proceedings 1*, pages 305–319. Springer, 1998.

[11] Monica Nogueira, Marcello Balduccini, Michael Gelfond, Richard Watson, and Matthew Barry. An a-prolog decision support system for the space shuttle. In *Practical Aspects of Declarative Languages: Third International Symposium, PADL 2001 Las Vegas, Nevada, March 11–12, 2001 Proceedings 3*, pages 169–183. Springer, 2001.

[12] Axel Polleres. From sparql to rules (and back). In *Proceedings of the 16th international conference on World Wide Web*, pages 787–796, 2007.

[13] Elena Mastria, Jessica Zangari, Simona Perri, and Francesco Calimeri. A machine learning guided rewriting approach for asp logic programs. *arXiv preprint arXiv:2009.10252*, 2020.

[14] Georg Henrik Von Wright. An essay in modal logic. 1951.

[15] Kaarlo Jaakko Juhani Hintikka. Knowledge and belief: An introduction to the logic of the two notions. 1962.

[16] Saul A Kripke. A completeness theorem in modal logic1. *The journal of symbolic logic*, 24(1):1–14, 1959.

[17] Saul A Kripke. Semantical analysis of modal logic i normal modal propositional calculi. *Mathematical Logic Quarterly*, 9(5-6):67–96, 1963.

[18] Saul A Kripke. Semantical analysis of modal logic ii. non-normal modal propositional calculi. In *The theory of models*, pages 206–220. Elsevier, 2014.

[19] Daniel Lehmann. Knowledge, common knowledge and related puzzles (extended summary). In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 62–67, 1984.

[20] Jorge Fandinno, Wolfgang Faber, and Michael Gelfond. Thirty years of epistemic specifications. *Theory and Practice of Logic Programming*, 22(6):1043–1083, 2022.

[21] Patrick Kahl, Richard Watson, Evgenii Balai, Michael Gelfond, and Yuanlin Zhang. The language of epistemic specifications (refined) including a prototype solver. *Journal of Logic and Computation*, 30(4):953–989, 2020.

[22] Pedro Cabalar, Jorge Fandinno, and Fariñas del Cerro Luis. Founded world views with autoepistemic equilibrium logic. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 134–147. Springer, 2019.

[23] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot asp solving with clingo. *Theory and Practice of Logic Programming*, 19(1):27–82, 2019.

[24] Pedro Cabalar, Jorge Fandinno, and Luis Fariñas del Cerro. Autoepistemic answer set programming. *Artificial Intelligence*, 289:103382, 2020.

[25] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Declarative problem-solving using the dlv system. *Logic-based artificial intelligence*, pages 79–103, 2000.

[26] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. gringo, clasp, clingo, and iclingo. 2010.

[27] Anthony P Leclerc and Patrick Thor Kahl. A survey of advances in epistemic logic program solvers. *arXiv preprint arXiv:1809.07141*, 2018.

[28] Michael Kelly. Wviews: A worldview solver for epistemic logic programs. *Honour's thesis, University of Western Sydney*, 2007.

[29] Tran Cao Son, Tiep Le, Patrick Thor Kahl, and Anthony P Leclerc. On computing world views of epistemic logic programs. In *IJCAI*, pages 1269–1275, 2017.

[30] Evgenii Balai and Patrick Kahl. Epistemic logic programs with sorts. *ASPOCP*, 2014, 2014.

[31] Wolfgang Faber and Michael Morak. Evaluating epistemic logic programs via answer set programming with quantifiers. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 6322–6329, 2023.

[32] Jonas Barklund. *Metaprogramming in logic*. Citeseer, 1994.

[33] Kenneth A Bowen. Amalgamating language and metalanguage in logic programming. *Logic programming*, 1982.

[34] Roland Kaminski, Javier Romero, Torsten Schaub, and Philipp Wanko. How to build your own asp-based system?! *Theory and Practice of Logic Programming*, 23(1):299–361, 2023.

[35] Thomas Eiter and Axel Polleres. Towards automated integration of guess and check programs in answer set programming: a meta-interpreter and applications. *Theory and Practice of Logic Programming*, 6(1-2):23–60, 2006.

[36] Martin Gebser, Jörg Pührer, Torsten Schaub, and Hans Tompits. A meta-programming technique for debugging answer-set programs. In *AAAI*, volume 8, pages 448–453, 2008.

[37] Gerhard Brewka, James Delgrande, Javier Romero, and Torsten Schaub. Implementing preferences with asprin. In *Logic Programming and Nonmonotonic Reasoning: 13th International Conference, LPNMR 2015, Lexington, KY, USA, September 27-30, 2015. Proceedings 13*, pages 158–172. Springer, 2015.

[38] Yannis Dimopoulos, Martin Gebser, Patrick Lühne, Javier Romero, and Torsten Schaub. plasp 3: Towards effective asp planning. *Theory and Practice of Logic Programming*, 19(3):477–504, 2019.

[39] Richard W Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. *Artificial intelligence*, 13(1-2):133–170, 1980.

[40] Kenneth A Bowen and Tobias Weinberg. A meta-level extension of prolog. 1985.

[41] Hamid Bacha. Meta-level programming: a compiled approach. 1987.

[42] Paul Broome and James Lipton. Combinatory logic programming: Computing in relation calculi. In *ILPS*, volume 94, pages 269–285, 1994.

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

[44] Il Moon. Modeling programmable logic controllers for logic verification. *IEEE Control Systems Magazine*, 14(2):53–59, 1994.

[45] Gottlob Frege et al. Begriffsschrift, a formula language, modeled upon that of arithmetic, for pure thought. *From Frege to Gödel: A source book in mathematical logic*, 1931:1–82, 1879.

[46] John McCarthy. Programs with common sense, 1959.

[47] Potassco. Clingo documentation. Web, 2020. Clingo AST page.

[48] Wolfgang Faber, Michael Morak, and Lukáš Chrpa. Determining action reversibility in strips using answer set and epistemic logic programming. *Theory and Practice of Logic Programming*, 21(5):646–662, 2021.

[49] Han Reichgelt. A review of mcdermott's "critique of pure reason". *AI Communications*, (1):39–42, 1987.

[50] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. A logic programming approach to knowledge-state planning, ii: The dlvk system. *Artificial Intelligence*, 144(1-2):157–211, 2003.

[51] Steve Hanks and Drew McDermott. Nonmonotonic logic and temporal projection. *Artificial Intelligence*, 33(3):379–412, 1987.

[52] Martin Brain, James H Davenport, and Alberto Griggio. Benchmarking solvers, sat-style. In SC^2 *@ ISSAC*, 2017.

[53] Martin C Cooper, Andreas Herzig, Faustine Maffre, Frédéric Maris, Elise Perrotin, and Pierre Régnier. A lightweight epistemic logic and its application to planning. *Artificial Intelligence*, 298:103437, 2021.

9 Appendix A

9.1 Generator Meta-Programs

Common Optimization Program

```
symbolic_atom(SA) :- atom_map(SA, _).

symbolic_epistemic_atom(k(A)) :- symbolic_atom(k(A)).
symbolic_objective_atom(OSA)  :- symbolic_atom(OSA),
                                 not symbolic_epistemic_atom(OSA).

epistemic_atom_map(KSA, KA) :- atom_map(KSA, KA),
                               symbolic_epistemic_atom(KSA).
objective_atom_map(OSA, OA) :- atom_map(OSA, OA),
                               symbolic_objective_atom(OSA).

epistemic_atom_int(KA) :- epistemic_atom_map(_, KA).
objective_atom_int(A)  :- objective_atom_map(_, A).

epistemic_map(KA,OA) :- epistemic_atom_map(KSA, KA),
                        objective_atom_map(OSA, OA), KSA = k(OSA).
```

Fact Optimization Program

```
:- fact(OSA), epistemic_atom_map(k(OSA), KA), not hold(KA).

positive_extra_assumptions(OSA) :- fact(OSA),
                                   symbolic_epistemic_atom(k(OSA)).

#show positive_extra_assumptions/1.
```

Preprocessing Optimization Program

```
:- cautious(SA), atom_map(SA, A), not hold(A).
positive_extra_assumptions(OSA) :- cautious(OSA),
                                   symbolic_epistemic_atom(k(OSA)).
#show positive_extra_assumptions/1.
```

Fact Propagation Program

```
:- kp_hold(OA), epistemic_map(KA, OA), not hold(KA).

kp_hold(OA) :- cautious(OSA),  objective_atom_map(OSA, OA).

kp_conjunction(B) :- literal_tuple(B),
                     kp_hold(A) : literal_tuple(B, A), A > 0,
```

```
                    not epistemic_atom_int(A);
                    hold(A) : literal_tuple(B,  A), A > 0,
                    epistemic_atom_int(A);
                    kp_not_hold(A) : literal_tuple(B, -A), A > 0,
                    not epistemic_atom_int(A);
                    not hold(A) : literal_tuple(B, -A), A > 0,
                    epistemic_atom_int(A).

kp_not_conjunction(B) :- literal_tuple(B), kp_not_hold(A),
                         literal_tuple(B,  A), A > 0,
                         not epistemic_atom_int(A).
kp_not_conjunction(B) :- literal_tuple(B), not hold(A),
                         literal_tuple(B,  A), A > 0,
                         epistemic_atom_int(A).
kp_not_conjunction(B) :- literal_tuple(B), kp_hold(A),
                         literal_tuple(B, -A), A > 0,
                         not epistemic_atom_int(A).
kp_not_conjunction(B) :- literal_tuple(B), hold(A),
                         literal_tuple(B, -A), A > 0,
                         epistemic_atom_int(A).

kp_body(normal(B))     :- rule(_, normal(B)), kp_conjunction(B).
kp_not_body(normal(B)) :- rule(_, normal(B)), kp_not_conjunction(B).

singleton_disjuntion(H) :- rule(disjunction(H), _),
                        #count{
                            A : atom_tuple(H, A)
                        } = 1.

kp_hold(A) : atom_tuple(H,A) :- rule(disjunction(H), B),
                                singleton_disjuntion(H), kp_body(B).

rule_head_tuple(H, B) :- rule(disjunction(H), B).
rule_head_tuple(H, B) :- rule(choice(H), B).

kp_not_hold(A) :- objective_atom_int(A),
                  kp_not_body(B) : atom_tuple(H,A),
                  rule_head_tuple(H, B).

zhold(SA)        :- hold(A), atom_map(SA, A).
z_kp_hold(SA)     :- kp_hold(A), atom_map(SA, A).
z_kp_not_hold(SA) :- kp_not_hold(A), atom_map(SA, A).
z_rule_head(disjunction(SA),rule(H,B)) :- rule(H, B),
                                          H = disjunction(H1),
                                          atom_tuple(H1,A), atom_map(SA, A).
```

```
z_rule_head(choice(SA),rule(H,B)) :- rule(H, B),
                                     H = choice(H1),
                                     atom_tuple(H1,A), atom_map(SA, A).
z_rule_body(normal(SA),rule(H,B)) :- rule(H, normal(B)),
                                     literal_tuple(B,A), A > 0, atom_map(SA, A).
z_rule_body(normal(-SA),rule(H,B)) :- rule(H, normal(B)),
                                      literal_tuple(B,-A), A > 0, atom_map(SA, A).

positive_extra_assumptions(OSA) :-
    kp_hold(OA),
    objective_atom_map(OSA,OA),
    symbolic_epistemic_atom(k(OSA)).
negative_extra_assumptions(OSA) :-
    kp_not_hold(OA),
    objective_atom_map(OSA,OA),
    symbolic_epistemic_atom(k(OSA)).
#show positive_extra_assumptions/1.
#show negative_extra_assumptions/1.

#external only_proved_candidates.
#external only_unproved_candidates.

explit_proven_candidates :- only_proved_candidates.
explit_proven_candidates :- only_unproved_candidates.

unproved(OA) :- explit_proven_candidates, epistemic_map(KA, OA),
                hold(KA), not kp_hold(OA).
exists_unproved :- explit_proven_candidates, unproved(_).

:-     exists_unproved, only_proved_candidates.
:- not exists_unproved, only_unproved_candidates.
```