arXiv:1401.6312v1 [cs.LO] 24 Jan 2014

Predicate Logic as a Modelling Language: The IDP System

Broes De Cat, Bart Bogaerts, Maurice Bruynooghe, and Marc Denecker

Department of Computer Science, KU Leuven

Abstract With the technology of the time, Kowalski's seminal 1974 paper *Predicate Logic as a Programming Language* was a breakthrough for the use of logic in computer science. The more recent tremendous progress in automated reasoning technologies, particularly in SAT solving and Constraint Programming, has paved the way for the use of logic as a modelling language. This paper describes the realisation of such a modelling language as the IDP knowledge-base system (KBS). In contrast to declarative programming, the user only specifies her knowledge about a problem and has not to pay attention to control issues. In the IDP system, declarative modelling is done in the language FO$(\cdot)^{\text{IDP}}$ which combines inductive definitions (similar to sets of Prolog rules) with first-order logic, types and aggregates, allowing for concise specifications.
The paper presents the language, motivates the design choices and gives an overview of the system architecture and the implementation techniques. It also gives an overview of different inference tasks supported by the system such as query evaluation, model expansion and theorem proving, and explains in detail how combining various functionalities results in a state-of-the-art model expansion engine. Finally, it explains how a tight integration with a procedural language (Lua) allows users to treat logical components as first-class citizens and to solve complex problems in a workflow of (multi-inference) interactions.

1 Introduction

Programming has evolved from writing assembler code, that directly addresses the hardware, to using high-level programming languages. The latter is made possible by a large number of automatic compilation steps before reaching that hardware level. With the advent of these higher level languages, more complex tasks, with large numbers of various constraints, are being addressed. Moreover, to accommodate for frequent changes in requirements, the ease of understanding and maintenance of software gain importance. Finally, to automate the many tasks that are currently still solved in a laboriously manual way, it is essential that solving these tasks requires less programming expertise and can more easily be mastered by domain experts. There is evidence of this evolution in many fields, including *verification* [18], *administration* [38], *scheduling* [5], *data mining* [8], *robotics* [59] and *configuration* [62].

The foundation of the field of Logic Programming (LP) in 1974 with Kowalski's seminal paper *Predicate Logic as a Programming Language* [45] was an important step in that direction, by giving the Horn-clause subset of predicate logic a procedural interpretation to use it for programming. Gradually, it emerged that a logic program is in fact a definition and that the well-founded semantics [61] is the most natural semantics to capture the meaning of definitions [20,21]. The XSB Prolog system [14] was the first to support the well-founded semantics. Since that time, tremendous progress has been made in automated reasoning technology, particularly in SAT solving and Constraint Programming (CP). This has allowed the field of LP to explore more pure forms of declarative programming, where control is handled by the solver and the user only has to care about the problem specification, and for which *declarative modelling* is perhaps a more appropriate term. The best-known exponent of this research is the field of Answer Set Programming (ASP) [6,11,34], where logic programs are interpreted according to the stable semantics.

Here we describe FO$(\cdot)^{\texttt{IDP}}$, a different approach which stays closer to the origins of logic programming. It integrates inductive definitions (a generalisation of Prolog's rules under the well-founded semantics) with First-Order Logic (FO) formulas to express general knowledge about the problem domain. The IDP system that supports the FO$(\cdot)^{\texttt{IDP}}$ language is conceived as a *Knowledge-Base System* (KBS). A KBS essentially consists of two components. On one hand, a *language* which is both formal (and unambiguous) and as natural as possible (i.e., the intended meaning of a sentence should correspond to its semantics) and, on the other hand, as many *inference techniques* as necessary. The more inference techniques are implemented in the KBS, the less programming is required of its users. The paradigm is inspired by several observations. First, *imperative programming* languages allow to directly encode specialised algorithms, but knowledge about the problem domain is hidden deep within those algorithms. This facilitates high-performance solutions, but makes debugging and maintenance very difficult. Second, a program is typically written to *perform one task* and perform it well, but cannot handle many related tasks based on the same knowledge. Third, *knowledge representation* languages excel at representing knowledge in a natural, human-understandable format. Programming language designers are starting to realize this and provide constructs to express generic knowledge, such as the LINQ data queries in C# and annotation-driven configuration. Lastly, the above-mentioned progress in automated reasoning techniques allows to move the control burden from programmer to inference engine ever more. The Knowledge-Base paradigm is an answer to these observation: application knowledge is modelled in a high-level Knowledge Representation (KR) language and state-of-the-art inferences techniques are applied to reason on the modelled knowledge. It has also been demonstrated that, while the KBS approach cannot compete with highly tuned algorithms, the effort to reach an acceptable solution (w.r.t. computing time or solution optimality) can be much smaller than that to develop an algorithmic solution [34,8]. Furthermore, the declarative approach is less error-prone and more maintainable.

In this paper, we present the IDP system, a state-of-the art KBS. The system already exists for several years, but only recently evolved into a KBS. Up until 2012, IDP was a model expansion system (the IDP^2 system)[1] capable of representing knowledge in a rich extension of FO and performing model expansion by applying its grounder GIDL and its solver MINISAT(ID). Recently, we have extended it into *the* IDP *knowledge-base framework* for general knowledge representation and reasoning (referred to as IDP^3); the earlier technology is reused for its model expansion inference. The IDP system goes beyond the KBS paradigm: for a KBS to be truly applicable in practical software engineering domains, it needs to provide an imperative programming interface, see [55]. Such an interface, in which logical components are first-class citizens, allows users to handle in- and output (e.g., to a physical database), to modify logical objects in a procedural way and to combine multiple inferences to solve more involved tasks. In this paper, we use KBS to refer to this 3-component architecture consisting of language, inferences and procedural integration. The IDP system provides such a procedural integration through the scripting language Lua [41]. The system's name IDP, *Imperative-Declarative Programming*, also refers to this paradigm.

In the work revolving around IDP, we can distinguish between the knowledge representation language FO$(\cdot)^{\texttt{IDP}}$ and the state-of-the-art inference engines. One can *naturally* model diverse application domains in FO$(\cdot)^{\texttt{IDP}}$; this contrasts with many approaches that *encode* knowledge such that a specific inference task becomes efficient. Furthermore, *reuse* of knowledge is central. FO$(\cdot)^{\texttt{IDP}}$ is modular and provides fine-grained management of logic components. E.g., it supports *namespaces*: formulas and terms can be declared in one component and used in several other components. The implementation of the inference engines provided by IDP aim at the reuse of similar functionality (see Section 6). This has two important advantages: (i) improvement of one inference engine (e.g., due to progress in one field of research) immediately has a beneficial effect on other engines; (ii) once "generic" functionality is available, it becomes easy to add new inference engines. To lower the bar for modellers, we aim at reducing the importance of clever modelling on the performance of the inference engines. Several techniques, such as grounding with bounds [66], function detection [16], automated translation to the most suitable solving paradigm [15] and automated symmetry breaking [26] have been devised to reduce the need for an expert modeller.

The rest of the paper is structured as follows. In Section 2, we present the syntax and semantics of FO(ID, Agg, PF), the logic underlying the system. The general IDP framework as well as a running example about course administration are presented in Section 3. This is followed by the main KBS components; the knowledge representation language FO$(\cdot)^{\texttt{IDP}}$ is described in Section 4 and the system's high-level architecture and main inference engines in Section 5. In Section 6, we provide a detailed workflow of the state-of-the-art model expansion inference engine. Applications, tools and modelling methodologies are discussed

[1] Given a logical theory and a structure interpreting the domain of discourse, model expansion searches for a model of the theory that extends the structure.

in Section 7, followed by related work and a conclusion. In the rest of the paper, we use IDP to refer to the current (2013), knowledge-base version of the system.

2 Formal base language

Before defining the language of the IDP system, we give an overview of its formal basis, the logic FO(ID, Agg, PF), an extension of FO with inductive definitions, aggregates and partial functions.

A vocabulary Σ consists of a set of predicate and function symbols, each with an associated *arity*, the number of arguments they take. We sometimes use P/n (f/n) to denote the predicate symbol P (respectively function symbol f) with arity n. A function can be declared as *partial*, indicating that for some inputs, the output can be undefined; otherwise it is *total*. Every vocabulary contains comparison operators: the predicates $= /2$, $\neq /2$, $< /2$, $> /2$, $\geq /2$ and $\leq /2$.

A *term* is a domain element, a variable or an n-ary function symbol applied to n terms. An *atom* is an n-ary predicate symbol applied to n terms. An atom is a *formula*; if φ and φ' are formulas, then $\neg\varphi$, $\varphi \wedge \varphi'$, $\varphi \vee \varphi'$, $\forall x : \varphi$ and $\exists x : \varphi$ are also formulas. The expressions $\varphi \Rightarrow \varphi'$, $\varphi \Leftarrow \varphi'$ and $\varphi \Leftrightarrow \varphi'$ are (as usual) shorthands for $\neg\varphi \vee \varphi'$, respectively $\varphi \vee \neg\varphi'$ and $(\neg\varphi \vee \varphi') \wedge (\varphi \vee \neg\varphi')$. A *literal* l is an atom a or its negation $\neg a$. A *sentence* is a formula without free (unquantified) variables. A *theory* $\mathcal{T}$ over a vocabulary Σ consists of a set of sentences with symbols in Σ. The vocabulary of a theory $\mathcal{T}$ is denoted voc($\mathcal{T}$). A term t containing occurrences of a term t' is denoted as $t[t']$; the replacement of t' in t by t'' is denoted as $t[t'/t'']$ (similarly for formulas).

A structure over a vocabulary Σ consists of a *domain* D (a totally ordered set), and a (partial) Σ-interpretation. A (partial) Σ-interpretation $\mathcal{I}$ is an interpretation for all symbols in Σ; we use $s^{\mathcal{I}}$ to refer to the interpretation of a symbol s in $\mathcal{I}$. An interpretation of a predicate symbol P/n consists of two subsets of D^n, denoted as $P^{\mathcal{I}}_{ct}$ and $P^{\mathcal{I}}_{cf}$; a (partial) interpretation for a function symbol f/n is a mapping $D^n \to 2^D$. If $f(\overline{d})^{\mathcal{I}} = \emptyset$, we say the image is *undefined*. The interpretation of a term with a direct subterm that is undefined is also undefined; that of an atom with a direct subterm that is undefined is *false*. Motivation for these choices is given in the next section. The interpretation $\{P^{\mathcal{I}}_{ct}, P^{\mathcal{I}}_{cf}\}$ of a predicate symbol P is *consistent* if $P^{\mathcal{I}}_{ct}$ and $P^{\mathcal{I}}_{cf}$ are disjoint. The interpretation of a partial function is always consistent; that of a (total) function is consistent when no tuple has the empty set as image. A Σ-interpretation is consistent if the interpretations of all its symbols are consistent. By abuse of notation, we will often identify a structure with its Σ-interpretation if the domain is clear from the context.

To support arithmetic operations and aggregates, the domain of every structure is a superset of all integer and real numbers. They are ordered before every other domain element, and internally they follow the natural order.

An interpretation for a predicate symbol P/n is two-valued if it is consistent and $P^{\mathcal{I}}_{ct} \cup P^{\mathcal{I}}_{cf} = D^n$; an interpretation for a total n-ary function symbol is two-valued if all images (over D^n) are singletons; for a partial function, it is

two-valued if all images are either singletons or empty sets. An interpretation is two-valued if the interpretations of all its symbols are. In two-valued structures $\mathcal{I}$, the *value* of terms and formulas without free variables is defined as follows. The value of a term t, denoted as $t^{\mathcal{I}}$, is d if t is the domain element d or if t is of the form $f(\bar{t})$ with $d \in f^{\mathcal{I}}(\bar{t}^{\mathcal{I}})$. The value of an atom $P(\bar{t})$ is true if $\bar{t}^{\mathcal{I}} \in P^{\mathcal{I}}_{ct}$ and false otherwise. The value of $\varphi \wedge \varphi'$ is true if both φ and φ' are true, $\varphi \vee \varphi'$ if at least one is true, $\neg\varphi$ if φ is false; $\forall x : \varphi$ is true if $\varphi[x/d]$ is true for all $d \in D$, $\exists x : \varphi$ if $\varphi[x/d]$ is true for at least one $d \in D$. We then say a two-valued structure M is a *model* of a formula φ or M *satisfies* φ, denoted as $M \models \varphi$, if φ is true in M. The comparison operators are interpreted as usual according to the total order on D. Given two tuples $\bar{t}$ and $\bar{t}'$ of terms of equal length n, $\bar{t} = \bar{t}'$ denotes the conjunction $t_1 = t'_1 \wedge \cdots \wedge t_n = t'_n$. For a vocabulary V and a structure S over $V_S \supseteq V$, $S|_V$ denotes the restriction of S to the symbols in V.

Unless the context specifies it differently, φ denotes a formula, t a term, D a domain, $\mathcal{I}$ a structure, d a domain element, x and y variables and $\sim$ any comparison operator.

2.1 Sets and Aggregates

Set expressions are expressions of the forms $\{\overline{x} : \varphi\}$ and $\{\overline{x} : \varphi : t\}$, with φ any formula and t any term. Given a domain D, an interpretation $\mathcal{I}$ and an assignment $\overline{d}$ to the free variables $\overline{y}$ of a set expression, the interpretation $\{\overline{x} : \varphi[\overline{y}/\overline{d}]\}^{\mathcal{I}}$ is the set $\{\overline{d}' \in \overline{D} \mid \varphi[\overline{x}/\overline{d}', \overline{y}/\overline{d}]^{\mathcal{I}} \text{ is } \mathbf{t}\}$, the interpretation of $\{\overline{x} : \varphi[\overline{y}/\overline{d}] : t[\overline{y}/\overline{d}]\}^{\mathcal{I}}$ is the multiset $\{(t[\overline{x}/\overline{d}', \overline{y}/\overline{d}]^{\mathcal{I}}) \mid \overline{d}' \in \overline{D} \text{ and } (\varphi \wedge \exists y' : t = y')[\overline{x}/\overline{d}', \overline{y}/\overline{d}]^{\mathcal{I}} \text{ is } \mathbf{t}\}$. Thus, in the context of a given assignment for the variables $\overline{y}$, the expression denotes the multiset of tuples t for which φ holds. Note that direct subterms of the set expression that are undefined are excluded from the set (the $\exists y' : t = y'$ subformula).

We extend the notion of term to include *aggregate terms*. Aggregate terms are of the form $agg(S)$, with S a set expression and agg an aggregate function. Currently, the aggregate functions cardinality, sum, product, minimum and maximum are defined. The last four only take sets with tuples of arity 1. The cardinality function maps a set interpretation to the number of elements it contains. The aggregate functions sum, product, minimum and maximum map a set to respectively the sum, product, minimum and maximum (on the order of the domain elements) of the first (and only) element in each tuple in the set. Sum and product are defined to be 0, respectively 1, if the set is empty. Note that sum and product aggregate terms are undefined if the set interpretation contains non-numeric values; minimum and maximum are undefined for the empty set (so are partial functions).

2.2 Definitions

We extend the notion of formulas to include *definitions*. Definitions Δ are sets of rules of the forms $\forall \overline{x} : P(\bar{t}) \leftarrow \varphi$ or $\forall \overline{x} : f(\bar{t}) = t' \leftarrow \varphi$, with the free variables

of φ amongst the $\overline{x}$. We refer to $P(\overline{t})$ and $f(\overline{t}) = t'$ as the *head* of the rule and to φ as the *body*. In the first form, P is the defined symbol; in the second, f is. The defined symbols of Δ are all symbols that are defined by at least one of its rules; all other symbols occurring in Δ are called *parameters* or *open* symbols of Δ. Intuitively, for each two-valued interpretation of the parameters, Δ determines the interpretation of the defined symbols in a unique way.

The satisfaction relation of FO can be extended to handle definitions by means of the well-founded semantics [60]. This semantics formalizes the informal semantics of rule sets as inductive definitions [20,21]. First, consider definitions Δ that only define predicate symbols and let Δ' be the definition constructed from Δ by replacing each rule $\forall \overline{x} : P(\overline{t}) \leftarrow \varphi$ with $\forall \overline{y} : P(\overline{y}) \leftarrow \exists \overline{x} : \overline{t} = \overline{y} \wedge \varphi$. The interpretation $\mathcal{I}$ satisfies Δ ($\mathcal{I} \models \Delta$) if $\mathcal{I}$ is a parametrised well-founded model of Δ, that means that $\mathcal{I}$ is the well-founded model of Δ' when the open symbols are interpreted as in $\mathcal{I}$.

When functions are involved, we transform them away so that we can use parametrised well-founded models as above. Let the *graph* predicate symbol of a function f/n be the new predicate symbol $F/n + 1$; given an interpretation $\mathcal{I}$, let $\mathcal{I}$' be $\mathcal{I}$ with the interpretations of all function symbols f/n replaced by an interpretation of the corresponding graph predicate symbols $F/n + 1$ such that $F(\overline{d}, d')$ holds in $\mathcal{I}$' if and only if $f(\overline{d}) = d'$ holds in I. Let Δ' be the definition constructed from Δ by replacing rules defining predicate symbols as above and replacing rules $\forall \overline{x} : f(\overline{t}) = t' \leftarrow \varphi$ as follows: first, replace them by $\forall \overline{y}, y' : F(y_1, \ldots, y_n, y') \leftarrow \exists \overline{x} : \overline{t} = \overline{y} \wedge t' = y' \wedge \varphi$. Next, replace occurrences of terms $f(\overline{t})$ in the bodies. An atom $A[f(\overline{t})]$ is replaced by $\exists x : A[f(\overline{t})/x] \wedge F(\overline{t}, x)$; a set expression $\{\overline{x} : \varphi : f(\overline{t})\}$ is replaced by $\{\overline{x} : \varphi \wedge \exists x : F(\overline{t}, x) : x\}$; concerning priority of application, replacement is done from the leaves of the parse tree upwards. We say that $\mathcal{I}$ satisfies the definition Δ ($\mathcal{I} \models \Delta$) if $\mathcal{I}'$ is a parametrised well-founded model of definition Δ'.

The *completion* of Δ for a symbol P, defined in Δ by the rules $\forall \overline{x}_i : P(\overline{t}_i) \leftarrow \varphi_i$ with $i \in [1, n]$, is the set consisting of the sentence $\forall \overline{x}_i : \varphi_i \Rightarrow P(\overline{t}_i)$ for each $i \in [1, n]$ and the sentence $\forall \overline{x} : P(\overline{x}) => \bigvee_{i \in [1,n]} (\overline{x} = \overline{t}_i \wedge \varphi_i)$; the completion for defined function symbols is defined similarly. This set is denoted as $comp_{P,\Delta}$, the union of all these sets for Δ as $comp_\Delta$. It is well-known that if $I \models \Delta$ then $I \models \text{comp}_\Delta$ but not always vice-versa (e.g., the inductive definition expressing transitive closure is stronger than its completion).

2.3 Types

The above language FO(ID, Agg, PF) is already a rich KR language. However, in many real applications, the use of *types* might be desired. Here, we present a typed extension of this language in terms of a translation to FO(ID, Agg, PF).

A vocabulary not only consists of predicate and function symbols, but also contains type symbols. Furthermore it assigns to each of its predicate and function symbols, a tuple of types (of the correct arity). Intuitively, it means that these predicates and functions are only defined on the (tuples of) domain elements within the correct type. Note that, for functions, totality means "defined

over every well-typed tuple" instead of over the whole universe as in an untyped logic. Lastly, every quantified variable is also typed, meaning that the quantification does not range over the entire domain, but only over that specific type.

Semantics of the typed language are defined in terms of the untyped language: types are treated as unary predicate symbols. In every structure, the interpretation of a typed predicate symbol can only contain tuples of domain elements within the correct type (for which the corresponding unary predicate symbol is true) and functions are only defined on those elements within the correct type. Note that this means that the translation of the typed language to the untyped language makes all function partial. Every quantification $\forall x : \varphi$, $\exists x : \varphi$, $\{x \mid \varphi\}$ or $\forall x : P(t) \leftarrow \varphi$, where x is typed T, is replaced by (respectively) $\forall x : T(x) \Rightarrow \varphi$, $\exists x : T(x) \wedge \varphi$, $\{x \mid T(x) \wedge \varphi\}$ or $\forall x : P(t) \leftarrow T(x) \wedge \varphi$.

Types are intended to mimic the classification of the domain of discourse. As such, it is natural to extend the notion of type to type *hierarchies*: a type can be a subtype or supertype of another type, indicating that an interpretation of the former will be a subset/superset of the latter.

3 IDP as a KBS

We start the section with a description of the architecture and a discussion of design decisions. We finish with sketching an application where the same knowledge is used for different tasks.

3.1 Architecture and design decisions

Here, we introduce the basic design decisions underlying the IDP system, the decisions that determine the look and feel of IDP as a KBS. While the implementation and the algorithms used in it may vary over time, these decisions are rather fixed. Let us recall the architecture of the KBS. Besides the two main components, the *language* and the *inferences*, there is also a *procedural integration* component. An overview is shown in Figure 1.

The first design decision, the one most visible to users, is about the language of the KBS. The language should be (i) *rich* enough so that users can express all their needs; (ii) *natural* enough so that theories stay close to the original (natural language) problem statement and are easy to read and to debug; and (iii) *modular* enough to allow for reuse and future extensions.

It is sometimes argued that the expressivity of a language should be limited, since this might "make the language" undecidable or intractable. We disagree. First, note that decidability and tractability depend on the task at hand. While deduction in first order logic is undecidable, other forms of inference, such as model expansion and querying in the context of a finite domain, are decidable. Second, while a more expressive language might allow users to express tasks high in the polynomial hierarchy, that does not imply that simple tasks become harder to solve. Rather to the contrary, stating the problem in a richer language

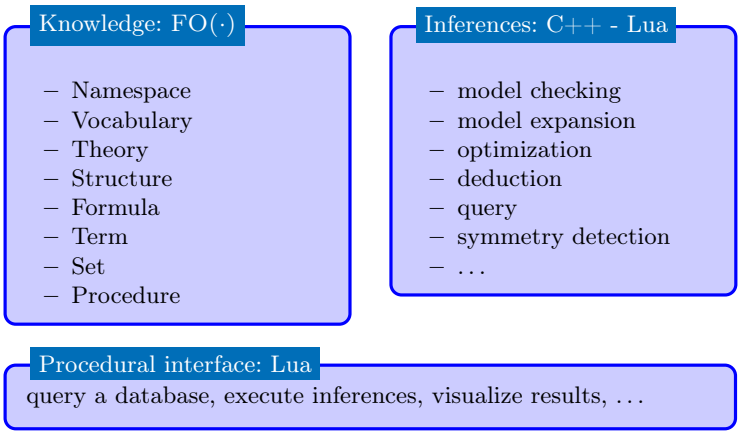

Figure 1. High-level representation of a KBS system

sometimes allow the KBS to exploit structural information that would be hidden in a more lower level problem statement.

To address the requirement of a rich and a natural language, we have opted for typed FO(ID, Agg, PF), FO extended with definitions, aggregates, partial functions and types. First order logic because conjunction, disjunction, universal and existential quantification have a very natural meaning. Extensions because FO has various weaknesses. Inductive definitions overcome the weakness that FO cannot express inductively defined concepts. Also non-inductive definitions are very useful. They often eliminate the use of ambiguous if statements (in mathematical texts, it is common practice to use "if" when defining concepts; this "if" actually corresponds to an equivalence in first-order logic, or, as we would say, a definitional implication). Aggregates allow users to concisely express information requiring lengthy and complex FO formulas. Types are omnipresent in the context of natural language, where quantification typically refers to a specific set of objects (e.g., everyone is mortal). For the integration with a procedural language, we did not pin ourselves to one specific language, currently offering an interface to the languages Lua and C++.

The third requirement, modularity, is important both at the language and at the systems level. Our choice for first order logic as base language implies, as it is sentence-based, that we can add language extensions without much interference at the syntactical level. Consider for example the introduction of aggregates above; we only need to extend satisfaction to atoms in which an aggregate occurs in order to obtain a semantics for a language extended with aggregates. Opting for a purely declarative language for representing knowledge is also crucial to the modularity of the system. It paves the way for using the same knowledge for different tasks. While tasks do have a procedural component, they are organised from the interface where particular inference methods are invoked on specific, purely declarative, theories and structures. New inference engines can be invoked

from that interface as they are added to the system. We have also attempted to organise inference engines in a modular way so that components can be reused in multiple engines. For example, the model expansion inference is currently implemented as ground-and-solve; the solver can be used separately from the grounder, and the grounding phase is composed of several smaller, reusable parts (such as, for example, evaluation of input-*-definitions [43]). Also various approaches to preprocess naive models are integrated in the system. Examples are symmetry detection and - breaking methods [26] and function detection methods [16].

Such preprocessing techniques also work towards the aim of the system to provide *robust* inference engines. Indeed, to separate modelling as much as possible from performance considerations of specific inference engines, techniques to detect and exploit implicit knowledge to increase efficiency are paramount.

3.2 Multiple inferences within one application domain

Given any knowledge base, there are often multiple applications that require different kinds of inference. By way of example, we explore the setting of a university course-management system. Its input is a database with information on students, professors, classrooms,.... One task of the system is to help students to choose their courses satisfying certain restrictions. Such an application is usually interactive; students make choices and, in between, the system checks the knowledge base. It removes choices when they become invalid, adds required prerequisites when a course is selected, ...; this is an example of *propagation* inference. Another task is to generate a schedule where every course is assigned a location and a starting time such that 1) no person has to be at two places at the same time, 2) no room is double-booked and 3) availability of professors is taken into account. Such an inference, searching for a valid solution, is called *model generation* or *model expansion*: one starts with partial information (availability of the professors, courses chosen by students, ...) and wants to extend it into a complete solution, namely a model of the scheduling theory. However, due to the large number of optional courses, such a solution (in which no student has overlapping courses) will probably not exist. In this case, we might want to find a solution in which the number of conflicts is minimal; this requires *minimisation* inference. Now, one might want to mail students with schedules with overlaps to give them the opportunity to change their selection. Hence, the solution of the minimisation inference should be *queried* to find the overlapping courses for every student. In the course of a semester, professors might have to cancel a lecture due to other urgent obligations. In that case, we want to find a *revision* of the current schedule, taking the changed restrictions into account and minimizing the number of changes with respect to the current schedule. In case such revisions are done manually, the *model checking* inference can be used so that no new conflicts are introduced. If some conflict does occur, an *explanation* should be provided. Finally, if a valid schedule is found, a *visualisation* inference can be used to create an easy-to-understand, visual representation of the schedule, personalised by the viewer's status (student, professor, administrative personnel, ...).

4 The Knowledge-base Language FO(·)$^{\text{IDP}}$

The language FO(·)$^{\text{IDP}}$, the knowledge representation language of the IDP system, is a modular and extendable language that provides a computer-readable notation for the concepts in FO(ID, Agg, PF), the formal base language.

In this section, we present the full FO(·)$^{\text{IDP}}$ language. The first part follows the structure of the previous section; it defines syntax of language components and links them to the formal concepts previously introduced. Next, we present extensions to the language that aim at increasing its capability to act as a knowledge representation language and/or its capability for use as a general programming paradigm. We then extend the language with features that provide modularity, such as namespaces and reuse of specifications. Finally, we show how to integrate it with a procedural language. A discussion on related languages is delayed until Section 8.

4.1 Language basics

Tokens and their role. Statements are terminated by "`;`". A tuple is specified by an enumeration of elements separated by commas and surrounded by brackets (e.g., "`(1,2)`"), a (multi)-set by an enumeration of elements separated by "`;`" and surrounded by curly brackets (e.g., "`{(1,2);(2,3);}`").

By *character*, we mean Latin letters, digits and most common special symbols ("`,`", "`.`", "`;`", ...); or formally, any ASCII-character in the range $32 - 127$ (note that this excludes, e.g., escape characters). A *string* is any sequence of characters except double quotes "`"`". A *name* is a string that starts with a Latin letter and may contain Latin letters, digits and the special characters "`'`" and "`_`". An *identifier* is either an integer, a floating point number ("`n.m`" with n an integer and m a natural number), a name or a string surrounded by quotes (e.g., "`"200 A"`"). Upper- and lowercase names are different identifiers[2]. Everything after the characters "`//`" and on the same line is considered a comment, as is everything between "`/*`" and the nearest next "`*/`".

Symbol declarations FO(·)$^{\text{IDP}}$ is a typed logic; the domain of discourse is divided (by the modeller) into sets of domain elements that conceptually belong to different types (or classes). As such, it has all the benefits of strongly-typed programming languages (Milner's "well-typed programs can't go wrong") at the expense of some verbosity. In the course management example, among the types considered are `courses`, `persons` and `locations`. Arguments of predicate and function symbols, as well as the the image of function symbols have to be typed by means of declarations. For example, the argument of a function `age` can be restricted to domain elements of type `person`, and the image to the natural numbers.

A vocabulary introduces a set of type, predicate and function symbols, as in the following example:

[2] The language does not use cases to distinguish between different kinds of identifiers.

```
vocabulary database is {
   type course;
   type person;
   type student subtype of person;
   pred takes[student, course];
   func age[person -> nat];
    ...
};
```

The vocabulary is named *database* and introduces three types; the third type, `student`, is declared as a subtype of `person` (all students are persons). It also declares a (binary) predicate symbol `takes`, a relation between students and courses, and a (unary) function symbol `age` mapping persons to their age (a natural number). There is no restriction on the order of the elements in a vocabulary; however, cyclic dependencies are forbidden; e.g., "`type student subtype of person; type person supertype of student;`" is illegal as `student` depends on `person` and vice versa.

A type is declared when a set of domain elements is judged to be of sufficient interest for the application at hand. As explained in Section 2.3, types can be compiled away; unary predicates are introduced and this results in an untyped logic. So, a type declaration `type T;` introduces in fact a unary predicate `T`. This translation is exploited in the language and one can use `T` both as type (e.g., in declarations of functions and predicates, as type of a quantified variable) and as predicate (e.g., to check that a domain element belongs to a subtype). In the other direction, a unary predicate does not introduce a type unless it is explicitly requested as in "`pred student[person]; student isa type;`". As we will show in Section 4.5, this creates room for more modularity in specifications. The type hierarchy is declared by subtype expressions (of the form "`T_1 subtype of T_2 ;`") and supertype expressions (of the form respectively "`T_1 supertype of T_2 ;`").

Well-typedness. A vocabulary is *well-typed* if the type hierarchy is acyclic. A theory is well-typed if all variables are typed and a term t of type T only occurs in positions that are typed as T or an ancestor of T. A structure is well-typed if no domain element d occurs in a position typed as T where $d \notin T$ and if for any two types T and T' with T declared as supertype of T' (or T' as subtype of T), $T' \subseteq T$ holds.

One advantage of guaranteeing well-typedness is that types with a common ancestor can safely share domain elements. Consider for example a type `car` that is interpreted by a set of identifiers for the different cars. Accidental use of a term typed `car` in an arithmetic operation will not be well-typed, as `car` is not a subtype of `int`. However, a term typed car can be used in an argument position declared as `vehicle` (with `vehicle` a supertype of `car`).

In Section 4.5, we show how overloading is supported.

Several shorthands are supported to reduce the verbosity of the language. For symbols of arity 0, one can write "`pred P;`" instead of "`pred P [];`" and "`func C->T;`" instead of "`func C[->T];`".

Structure. Structures are described by a list of equalities of symbols to sets of tuples, and can be partial.

```
structure data1 over database is {
   course = {Logic; Chinese};
   student = {1..3};
   takes :: ct = {1,Logic};
   takes :: cf = {2,Chinese};
   age :: ct = {1,25; 3,30};
};
```

declares a structure named "data1"; it interprets the type `course` by the domain elements "Logic" and "Chinese" and the type student by the identifiers 1, 2, and 3. It also states that the atom `takes(1,Logic)` is true (the "ct", or certainly true, table) and the atom `takes(2,Chinese)` false (the "cf" table). Note that `takes(1, Chinese)` is in none of the sets, so the structure only partially interprets "takes". Finally, the structure states that students `1` and `3` are respectively 25 and 30 years old, while the age of student `2` is not known in `data1`.

As the example shows, the general form of an interpretation statement has the form of `pX = S`, with `p` the symbol, `X` either empty (the interpretation is total), `::ct` (the true tuples are given) or `::cf` (the false tuples are given) and `S` a set of tuples. The set `S` can take different forms. First, it can be a plain enumeration of tuples of domain elements. Second, it can be an expression of the form `{n..m}`, with `n` and `m` either both integer numbers, floating-point numbers or chars and with $n \leq m$, expressing that $S = \{\langle x \rangle \mid x \in [n..m]\}$. Lastly, one can also express a procedural interpretation; this is discussed in Section 5.3.

Some symbols are builtin; by default they are part of every vocabulary and their interpretation is included in every structure. First, the binary comparison operators $=, \neq$ (written `~=`), $<, >, \leq$ (written `=<`), $\geq$ (written `>=`) can always be used. The interpretation of $=$ and $\neq$ is the standard one. For the last four, the total order over the domain elements determines the interpretation. Second, the 0-ary predicate symbols `true` (interpretation **t**) and `false` (interpretation **f**). Third, a number of pre-interpreted types are implicit in every vocabulary and interpreted in every structure: The set of all strings (`string`), the set of all Latin letters (`char`, subtype of `string`), and the sets of all natural, integer and real numbers (`nat`, `int` and `real` with `nat` subtype of `int` and `int` subtype of `real`) Fourth, a number of binary arithmetic functions is provided, namely addition `+`, subtraction `-`, multiplication `*`, division `/` and modulo `%`. All of these are declared over `real` and the last two are partial. Lastly, the functions *min*, *max*, *pred* and *succ* are defined for each (root) type in a vocabulary. In any structure, for type T, `min[->T]` and `max[->T]` map to the smallest, respectively largest, domain element in T (according to the total order on domain elements). The

partial functions `pred[T->T]` and `succ[T->T]` map domain elements of T to their predecessor, respectively successor, in T.

Theory. The following table shows how the basic operators of FO(ID, Agg, PF) are denoted in FO$(\cdot)^{\text{IDP}}$.[3]

FO$(\cdot)^{\text{IDP}}$	FO(ID, Agg, PF)	FO$(\cdot)^{\text{IDP}}$	FO(ID, Agg, PF)
`&`	$\wedge$	`|`	$\vee$
`=>`	$\Rightarrow$	`<=`	$\Leftarrow$
`<=>`	$\Leftrightarrow$	`<-`	$\leftarrow$
`!`	$\forall$	`?`	$\exists$
$\sim$	$\neg$		

Quantified formulas are written as "`!x in T: f`" or "`?x in T: f`" with `x` a variable, `T` a type and `f` a formula; a formula "`!x in T1, ..., xn in Tn: f`" is a shorthand for "`!x in T1: ... : !xn in Tn: f`".

Below is a small scheduling theory using functions `courseOf[session->course]`, `planned[session->timeslot]`, and `teaches[course->person]` and predicates `available[person,timeslot]`, `takes[student,course]`, and `attends[student,session]`. The first sentence states that a teacher is available for all sessions of the courses he teaches; the second one that each student attends each session of all courses he takes.

```
theory sessionAssignment over scheduling is {
   !sess in session: available(teaches(courseOf(sess)),planned(sess));
   !stud in student, sess in session: takes(stud,courseOf(sess))
      => attends(stud,sess);
};
```

Whereas set notation was used in structures for enumerating interpretations of symbols, in theories it is used for multisets. An expression `{x1 in T1, ..., xn in Tn : f : (t1 ,..., tn) }` represents the multiset $\{\langle t_1(\overline{x}), \ldots, t_n(\overline{x})\rangle \,|x_1 \in T_1 \wedge \ldots \wedge x_n \in T_n \wedge f(\overline{x})\}$, i.e., the multiset of tuples `(t1 ,..., tn)` (of the specified types) for which the formula `f` holds. The last argument (tuple of terms) is optional; if absent, the multiset of empty tuples is taken.

Multisets can be used in different contexts. The aggregate functions minimum, maximum, sum, product and cardinality, denoted as `min`, respectively `max`, `sum`, `prod` and `#` or `count`, take a multiset as their only argument. An alternative for constraints on the cardinality of sets are *extended* existential quantification, e.g., instead of writing `#{stud in student: attends(stud,sess)} >= 5`, one can write `?>=5 stud in student: attends(stud,sess)`.

Anywhere a multiset can be used, a predicate symbol P can also be used, which then acts as a shorthand for $\{\overline{x} \mid P(\overline{x})\}$. E.g., the number of courses can be represented as `#(Course)`.

[3] To distinguish inequality and implication operators, note that the latter form arrows.

Generalized quantifiers The language supports *generalized* quantifiers through expressions of the form "`!(x11 ,..., x1n) ...( xm1 ,..., xmn) in S: form`", where the domain of each tuple of variables consists of all tuples in the interpretation of the set `S` (and similarly for existential quantification). Formulas can also be used as guards, in case of a single tuple of variables, as `!(x1 ,..., xn) sat form1 : form2`, where the domain of `(x1 ,..., xn)` are all instantiations for which `form1` is satisfied, i.e., true.

Generalized quantifiers help to avoid common (beginner) errors in modelling. For example knowledge of the form "teacher A only teaches sessions on Wednesday" is too often incorrectly modelled as `!s in session: teaches(A,course(s)) & day(s)=Wednesday` and can now be unambiguously expressed as `!s sat teaches(A,course(s)): day(s)=Wednesday`.

4.2 Partial functions

In standard logic, function symbols denote total functions. In practice, partial functions are unavoidable, e.g., a function $spouse[person \rightarrow person]$ is naturally undefined for singles and the arithmetic operation division is undefined for zero. Partial functions are declared as `partial func` instead of plain `func`. Assigning a proper semantics to non-denoting terms, however, may give rise to ambiguity. For instance, $White(Unicorn)$ can be interpreted as "if the unicorn exists it is white" or as "the unicorn exists and is white".

In [64] and [32], it has already been pointed out that many approaches exist to avoid ambiguity. The simplest solution is to restrict the syntax of formulas. One could, e.g., only allow terms of the form $f(t)$ in contexts where it is certain that $f(t)$ is defined. This option is often taken in mathematics, where terms like, e.g., $\frac{1}{0}$ are considered meaningless, but quantifications of the form $\forall x : x \neq 0 \Rightarrow \frac{1}{x} \neq 42$ are allowed as it is clear that the division $\frac{1}{x}$ will be defined for all relevant x. This idea has been implemented for example in the Rodin toolset for Event-B [2], where for every occurrence of a partial function, it should be provable that the function will only by applied to values in its domain. It was also shown in [64] that, in the context of a KBS, such an approach is too restrictive; it can, e.g., not be used when it is not known in advance for which input arguments a function will be undefined. Another approach, proposed by Kleene [44], falls back to a three-valued logic, in which undefinedness is made explicit. From a practical point of view, however, the semantics are counter-intuitive on some types of formulas, which is undesirable in a KBS. The approach taken in [64] is one that was first proposed in [56]. The value of an atom $P(t)$, where t is a term with undefined subterms, is either **t** or **f** depending on the context in which the atom appears. In a positive (negative) context, it is interpreted as **t**, respectively **f**. Intuitively, this choice maximizes the truth-value of the formula it occurs in.

After some years of experimenting, we selected a still different approach, which turned out to be flexible, intuitive and to allow for elegant modelling. As already mentioned in Section 2.2, the semantics is based on the conversion of the function to its graph predicate and boils down to replacing the atom $A[f(\bar{t})]$, that provides the context for the term, by the formula $\exists x : A[x] \wedge F(\bar{t}, x)$.

Or, intuitively, for an atom to be true, nested function applications have to be defined. Note that this corresponds to the semantics of $A[f(\bar{t})] \wedge \exists x : f(\bar{t}) = x$. After this translation, the requirements of the Rodin toolset (provability that functions are only applied to values in their domain) are satisfied. We made the choice that atoms with undefined terms are considered false. The result is a semantics that is close to the *relational semantics* proposed in [32]. For cases where the default choice does not result in the intended meaning, we provide the abbreviation `denotes(f(t))` for $\exists x : f(t) = x$, which the user can insert at a different location to suit her needs. For example, the atom `cost/number=<100` , with `cost` and `number` both function symbols, is false when `number` equals zero. If the user wants the atom to be true, she can use `~exists(cost/number ) | cost/number=<100`. Similarly, by default, an aggregate term is undefined if any of the term instantiations of its set is undefined; however, one can use `denotes t & f` instead of `f` in the formula of the aggregate to deviate from this default.

4.3 Definitions

The logic FO$(\cdot)^{\text{IDP}}$ contains a definition construct to express different kinds of definitions. This construct is one of the most original aspects of FO$(\cdot)^{\text{IDP}}$ and we explain it in somewhat more detail. For even more details we refer to [23].

Definitions are one of the important blocks that science is built with. Much knowledge of a human expert consists of definitions. It is well-known also that, in general, inductive/recursive definitions cannot be expressed in FO. Definitions belong to informal language; no formal rules exist how to write a definition but a frequently used linguistic convention is to express them as a set of informal rules. This convention is formalized in FO$(\cdot)^{\text{IDP}}$ where a formal definition is expressed using the optional keyword `define` followed by a set of rules of the form "`! x1 ... xm: P(t1 ,..., tn) <- f`" or "`! x1 ... xm: f(t1 ,..., tn)=t0 <- f`", with `ti` a term, `f` a formula and `P` a predicate symbol. Observe that the second rule defines a function symbol f.

Informal definitions have some extraordinary properties. Certainly those used in formal mathematical text strike us for the precision of their meaning. The formal semantics of FO$(\cdot)^{\text{IDP}}$ definitions carefully formalizes this meaning. Several types of informal definitions can be distinguished. Below, the three most common ones are illustrated with two formal and one informal definition: Definition 1 is a non-recursive one, Definition 2 is a monotone one, while the informal one, Definition 3 is by induction over a well-founded order or semi-order. The latter definition is by induction over the subformula order. Definitions over a well-founded order frequently contain non-monotone rules. For instance the rule defining $I \models \neg\alpha$ has a non-monotone condition $I \not\models \alpha$.

Definition 1. *The relation of non-overlapping sessions is defined as:*

```
define  {
    !s in session: noOverlap(s,s);
```

```
    !(s1)(s2) in session: noOverlap(s1,s2) <-
                          planned(s1)+length(s1)=<planned(s2);
    !(s1)(s2) in session: noOverlap(s1,s2) <-
                          planned(s2)+length(s2)=<planned(s1);
};
```

Definition 2. *The relation canTake, expressing which courses can be taken according to the course-dependency relationship depends, is defined as:*

```
define {
    !c1 in course: canTake(c1) <- !c2 in course: depends(c1,c2)
                                      => canTake(c2);
};
```

Definition 3. *The satisfaction relation $\models$ of propositional logic is defined by induction over the structure of formulas:*

- $I \models P$ *if* $P \in I$.
- $I \models \alpha \wedge \beta$ *if* $I \models \alpha$ *and* $I \models \beta$.
- $I \models \alpha \vee \beta$ *if* $I \models \alpha$ *or* $I \models \beta$ *(or both).*
- $I \models \neg\alpha$ *if* $I \not\models \alpha$.

The different sorts of definitions have different semantic properties. It is commonly assumed that the defined set is the least set that satisfies the rules of the definition. However, this is only true for monotone definitions, but not for nonmonotone definitions such as Definition 3: there is no least relation that satisfies its rules. Still, there is an explanation that applies to both [12]: the set defined by an inductive definition is the result of a construction process. The construction starts with the empty set, and proceeds by iteratively applying non-satisfied rules, till the set is saturated. In the case of monotone definitions, rules can be applied in any order; but in the case of definitions over a well-founded order, rule application must follow the specified order. This condition is necessary for the nonmonotone rules. If they would be applied too early, later rule applications may invalidate their condition. E.g., in the initial step of the construction of $\models$, when the relation is still empty, we could derive $I \models \neg\varphi$ for each φ, but the condition $I \not\models \varphi$ will in many cases later become invalidated. The role of the induction order is exactly to prevent such an untimely rule application. E.g., in Definition 3, one is not allowed to apply a rule to derive $\mathcal{I} \models \varphi$ as long as there are unsatisfied rules for deriving the satisfaction of subformulas of φ.

The problem we face in formalizing this idea for the semantics of $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ definitions, is that the syntax of $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ does not specify an explicit induction order for nonmonotone $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ definitions. Thus, the question is whether one can somehow "guess" the induction order. Indeed, if we look back at Definition 3, we see that the order is implicit in the structure of the rules: formulas in the head of rules are always higher in the induction order than those in the body.

This holds true in general. It should be possible then to design a mathematical procedure that somehow is capable to exploit this implicit structure.

In [24], this idea was elaborated. The induction process of an FO$(\cdot)^{\text{ID}}$ definition is formalized as a sequence of three-valued structures of increasing precision. Such a structure records what elements have been derived to be in the set, what elements have been derived to be out of the set, and which have not been derived yet. Using three-valued truth evaluation, one can then establish whether it is safe to apply a rule or not. All induction sequences can be proven to converge. In case the definition has the form of a logic program and the underlying structure is a Herbrand interpretation, the resulting process can be proven to converge to the well-known well-founded model of the program [61]. As such, the semantics of FO$(\cdot)^{\text{ID}}$ definitions is a generalization of the well-founded semantics, to arbitrary bodies, arbitrary structures and with parameters. This (extended) well-founded semantics provides a uniform formalization for the two most common forms of induction (monotone and over a well-founded order) and even for the less common form of iterated induction. Compared to other logics of iterated inductive definitions, e.g. IID [12], the contribution is that the order does not have to be expressed, as this can be very tedious.

quite remarkable that, while the community has worried for so long about the question of formal and informal semantics of logic programming, there is such a clear and precise informal counterpart to of logic programs be found in natural language. Why this view never took off?

4.4 Constructed types

element; however, for various applications, it is more natural to treat functions as open. Allowing for this is currently an important topic in, e.g., ASP research [7,48]. In our setting, the inverse problem presents itself. In FO, functions are considered open and can take any interpretation. However, sometimes it is more natural to fix the interpretation of ground terms over a function f to unique and different domain elements of a specific type. Consider for example the notion of "direction" in a spatial-related application (with one constant for each direction, different from all other such constants), or the different days of the week in our scheduling application. One way to impose this is to explicitly express the *unique names* and *domain closure* axioms (UNA, DCA).

As the UNA and DCA are quite cumbersome to encode, the language provides a more natural way to express this, constructing a typed, local Herbrand interpretation given a set of new *constructor* function symbols. The statement `type days constructed from Monday, Tuesday, ..., Sunday` declares a type (and predicate) `days` and 7 new constants (`Monday` etc.) that all map to the type days. Additionally, in any structure, each of those constants is interpreted by a new, anonymous domain element that is different from all other domain elements (UNA), and `days` is interpreted as those 7 domain elements (DCA). Non-constant function symbols are also supported in this construct, as well as recursion. Hence one can for example declare a (Prolog-like) list of integers as `type list constructed from nil, cons[int, list]` or a data type of binary trees of integers as `type`

`tree` constructed from `nil`, `t[tree,int,tree]`. For the former, an anonymous domain element is constructed (lazily) for `nil` and for each term `cons(i,d)` with `i` an `int` and d an (anonymous) domain element of type `list`. Note that there are an infinite number of domain elements of type `list`. Having declared lists, one can write definitions such as `head(cons(h,t))=h` and `tail(cons(h,t))=t`.

A reader might note that various language expressions, such as recursively constructed types and types such as `int` and `real`, can give rise to infinite domains. Currently, most inference engines in IDP only provide preliminary support to generally handle such domains. In the context of model expansion, Section 6.5 discusses several approaches employed to that end.

4.5 Improve usability

In the previous subsections, we already introduced various shorthands to facilitate the comfort of the user; here we introduce two additional features: namespaces —which increase the modularity of the language– and overloading. We end this subsection with showing how these two features pave the road for inclusion and reuse of various pieces of FO$(\cdot)^{\text{IDP}}$ code.

Namespaces. Define a *logic component* as either a vocabulary, a theory, a structure, a term, a formula or a set. Each logic component o is associated with a *type* (`vocabulary`, respectively `theory`, `structure`, `term`, `formula` and `set`) and a vocabulary (unless it is a vocabulary itself), referred to as $type(o)$, respectively $voc(o)$.

Logic components are associated with names in the context of a *namespace*, a new type of language construct that maps names to logic components and (other) namespaces. A namespace will act as a context for logic components, to allow overloading and increase modularity. For example, the following specification declares two namespaces, `A` and `B`; each one contains a (different) vocabulary `V`; moreover, `B` contains a term `t` over the latter vocabulary.

```
namespace A is {
   vocabulary V is { ... };
};
namespace B is {
   vocabulary V is { ... };
   term t over V is sum({x: P(x): t(x)});
};
```

Logic components and namespaces not declared in an explicit namespace are considered part of the implicitly defined namespace `global`.

As is clear already in the above example, names themselves do not have to be unique and we need to allow the user to uniquely identify logic components and namespaces. For each name in a namespace, we define the *Fully Qualified Name*(FQN) by induction as follows. The global namespace has `global` as FQN. Any name `name` in a namespace `n` has as FQN `fn::name`, with `fn` the FQN

of namespace n. For symbols, the type information is included in the FQN, as to allow overloading within one vocabulary: the FQN of a predicate symbol `s[T1,...,Tn]` in a vocabulary V, with FQN `fv`, is `fv::s[T1...Tn]`. It is defined similarly for function symbols. In any well-typed specification, all FQNs have to be unique, except for the FQN of namespaces. All namespaces with the same FQN are considered to be partial declarations of one combined namespace.

Using the FQN, we could declare a theory `T` in `B` but over the vocabulary `V` in `A` by `theory T over global::A::V is { ... }`.

Overloading and disambiguation. Always having to use the full FQN for any name would result in lengthy specifications, as does the the requirement that every variable is typed. To resolve this, IDP has a disambiguation component in its preprocessing that associates names to logical components. When the ambiguity cannot be resolved, the user is forced to provide more details (the context and/or the type signature). Users need a good understanding of the disambiguation strategy as they often rely upon it to be less verbose. Not only can users omit parts of the FQN, they can also omit the type of quantified variables in sentences and definitions; they can even omit the outer universal quantifications, in which case all unknown names, in positions where a term is expected, are considered universally quantified over the whole sentence or definition. All this omitted information is recovered by the disambiguation strategy as follows.

A *disambiguation* is an assignment of types to variables and of logical components to (occurrences of) names that satisfies the following constraints.

- The type of the logic component has to be consistent with where it occurs in the specification. E.g., for `theory T over V is ...`, `V` has to be a vocabulary.
- The type of a variable is the most specific type in the type hierarchy that is a supertype of the types of all occurrences of the variable.
- If a logic component o is assigned to a name *name*, there has to be an FQN fqn that refers to o such that *name* equals fqn or *name* is a suffix of fqn. For symbols, *name* can also be a suffix of fqn without its type specification or $name/n$, with n the arity of the symbol.
- For atoms $P(t_1, \ldots, t_n)$, the arity of the symbol s assigned to P has to be n and the output type of each term and the type of the associated argument position of s have to have a common supertype in the type hierarchy. A similar property has to hold for nested terms.

A specification is well-typed if only one disambiguation exists. If no disambiguations exist, an error is reported; if multiple exist, the user is asked to provide more detail.

For example, consider the predicate symbols `P[T1]`, `P[T2]` and the function symbol `P[T1->T2]`, all declared in a vocabulary `V`. Their FQN is respectively `global::V::P[T1]`, `global::V::P[T2]` and `global::V::P[T1->T2]`. In that context, the sentence `P(P(x))` has only one disambiguation, so is well-typed. Indeed, the inner `P` can only refer to a function symbol named P, so there is only one possible value. In that case, `x` has to take type `T1` and the outer `P` has to be a

predicate symbol of arity one with an argument of type `T2`, which also uniquely determines the logic component. The full specification of that sentence would then be `!x in T1: global::V::P[T2](global::V::P[T1->T2](x))`.

Inclusion of logic components. To facilitate reuse, a vocabulary `V` can contain `include name` statements, where `name` refers to a vocabulary. The (conceptual) effect is as if vocabulary `V` also contained all declarations in the latter vocabulary. A simple example:

```
vocabulary database is {
   type course;   type location;   type session;
   func nameOf[course->string];
};
vocabulary schedule is {
   include database;
   func assigned[session -> course];
   func assigned[session -> location];
};
```

As vocabulary `schedule` includes vocabulary `database`, it also contains all symbols in `database`, such as `session`, and can use them in its own declarations.

Next to vocabularies, include statements are also allowed in theories, structures and namespaces, with analogous effect. Inclusion can be more refined, e.g., `include database::nameOf` includes only the function symbol `nameOf[course->string]` and the type `course` from `database` (`string` is present by default); similarly, a statement `include data1::age;` in a structure component includes only the interpretation of `age` from structure `data1`. Note that include statements have to be stratified.

5 Inferences and system architecture

In this section, we discuss the architecture and the inference methods of the IDP system.

5.1 Main Inferences

The IDP system supports a range of functionalities to solve various inference and other logic-related tasks. Below, we specify a task through its input arguments $i_1, \ldots, i_n$, a precondition $Pre(i_1, \ldots, i_n)$, output arguments $o_1, \ldots, o_m$, and a postcondition $Post(i_1, \ldots, i_n, o_1, \ldots, o_m)$. This specifies a task as a partial (potentially non-deterministic) function from a space of input values to a space of output values. An *inference* task is a task that has logic objects (logic components, symbols, domain elements, ...) as input and output and where the pre- and postcondition depend on semantic properties of these (logically equivalent input has logically equivalent output). For example, the normalization of

a formula is considered a task, but not an inference task: an input formula φ is transformed into a logically equivalent formula ψ. In the following, we discuss the inference tasks currently supported by IDP.

Querying structures takes as input a structure $\mathcal{I}$ and a set expression $\{x \mid \varphi\}$ over a vocabulary V and returns the set $\{x \mid \varphi\}^{\mathcal{I}}$. The formula φ can only contain symbols that are two-valued in $\mathcal{I}$. However, for any symbol $P(\overline{T})$ in a vocabulary V, V also contains two "derived" symbols $P_{ct}(\overline{T})$ and $P_{cf}(\overline{T})$. In any structure $\mathcal{I}$ over V, P_{ct} and P_{cf} are interpreted in $\mathcal{I}$ as the set of all tuples of domain elements for which P is true, respectively false, in $\mathcal{I}$ (and similarly for functions, which are interpreted over their graph). Query is implemented by transforming the set expression into an extended First-Order Binary Decision Diagram and afterwards querying the BDD over $\mathcal{I}$, as described in [66].

Model expansion takes as input a partial structure $\mathcal{I}_{in}$ and an FO$(\cdot)^{\texttt{IDP}}$ theory $\mathcal{T}$, and produces models $\mathcal{I}$ of $\mathcal{T}$ that expand $\mathcal{I}_{in}$. **Optimization inference** takes also a term t as input, and produces models that have a minimal value for t. Both inference tasks are discussed in more detail in Section 6. **Model checking** is a special case of model expansion with $\mathcal{I}$ a two-valued structure interpreting $voc(\mathcal{T})$; it is implemented through model expansion.

Propagation takes as input a theory $\mathcal{T}$ and a partial structure $\mathcal{I}$, returning a more precise partial structure $\mathcal{I}_{out}$ that approximates all models of $\mathcal{T}$ that are expansions of $\mathcal{I}$ [65]. Equivalently, propagation deduces ground literals that hold in all models of $\mathcal{T}$ that expand $\mathcal{I}$. The system provides a complete propagation system (co-NP complete) and a sound but incomplete polynomial algorithm based on a lifted form of unit propagation. The latter proved very useful for building interactive knowledge-based configuration systems [62].

Deduction takes as input an FO$(\cdot)^{\texttt{IDP}}$ theory $\mathcal{T}$ and an FO theory $\mathcal{T}_{FO}$ and returns true if $\mathcal{T} \models \mathcal{T}_{FO}$. It is implemented by translating $\mathcal{T}$ into a (weaker) FO theory $\mathcal{T}_0$ and calling the theorem prover Spass [63] to check whether $\mathcal{T}_0 \models \mathcal{T}_{FO}$.

Δ-model expansion takes as input a definition Δ and a structure $\mathcal{I}_{in}$, interpreting all parameters of Δ, and returns the unique model $\mathcal{I}$ that expands $\mathcal{I}_{in}$. This task is an instance of model expansion, but is solved in IDP using different technology. The close relationship between definitions and logic programs under the well-founded semantics is exploited to translate Δ and $\mathcal{I}_{in}$ into a tabled Prolog program, after which XSB is used to compute $\mathcal{I}$. Taking an extra formula φ as input, with free variables $\overline{x}$, the same approach is used to solve the query φ with respect to Δ and $\mathcal{I}_{in}$ in a goal-oriented way [43].

Symmetry detection takes as input a theory $\mathcal{T}$ and a partial structure $\mathcal{I}$ and returns *symmetries* over $\mathcal{T}$ and $\mathcal{I}$ [26]. A symmetry is a function, say f, mapping structures to structures, such that, for any two-valued expansion J of $\mathcal{I}$, $J \models \mathcal{T}$ iff $f(J) \models \mathcal{T}$.

Next to inferences, the system also provides a large number of smaller-scale functionalities. An example are (model-preserving) normalization procedures, that take a theory $\mathcal{T}$ and (optionally) a structure $\mathcal{I}$ and transform $\mathcal{T}$ into a theory satisfying certain properties. Below, an overview is given of a well-known subset of the normalization procedures provided by IDP.

flatten($\mathcal{T}$) puts disjunction and conjunctions in left-associative form.
simplify($\mathcal{T}$,$\mathcal{I}$) replaces terms and formulas known in $\mathcal{I}$ by their interpretation.
push-negations($\mathcal{T}$) puts a theory in negation normal form.
push-quantifications($\mathcal{T}$) moves quantifications down if the associated variable does not occur in all subformulas.
shared-formula($\mathcal{T}$) replaces terms and formulas that occur multiple times in $\mathcal{T}$ (possibly with different variable substitutions), by a new symbol, defined by that term or formula.
nest-variables($\mathcal{T}$) uses atoms $x = t$, with x a variable and t a term, to replace x by t wherever allowed, possibly eliminating $x = t$ itself.
ground($\mathcal{T}$,$\mathcal{I}$) transform $\mathcal{T}$ into a ground theory $\mathcal{T}_g$ such that models of $\mathcal{T}_g$ can be mapped one-to-one to models of $\mathcal{T}$ expanding $\mathcal{I}$. The ground theory can be transformed into different language formats, through the option `stdoptions.language`, such as CNF (Clausal Normal Form), ECNF (the native format of MiniSAT(ID)) or ground ASP.

Next to FO$(\cdot)^{\text{IDP}}$ inferences, more domain-specific inferences are provided. One example is unsat-debugging: given a theory $\mathcal{T}$ and structure $\mathcal{I}$, $\mathcal{I} \not\models \mathcal{T}$, find a minimal subset of $\mathcal{T}$ that is unsatisfiable in $\mathcal{I}$. Unsat-debugging is a great aid for users to help pinpoint modelling errors. Through reuse of the above-mentioned inferences and functionality, it took minimal time to implement in IDP.

5.2 Internal representation.

The system offers the language FO$(\cdot)^{\text{IDP}}$ to the user, but internally, a more manageable representation is necessary. The main representation closely follows the structure of the FO(ID, Agg, PF) language: each component of the parse-tree is stored as a separate object with pointers to its children. E.g., a *theory*-object consists of a set of sentences, which can be retrieved and modified; a namespace provides a mapping from names to its subcomponents and a name resolution mechanism. The only exception is for interpretations which come in many different flavours (enumerated, symbolic, procedurally interpreted) and can be very large, so a naive enumeration would consume too much memory while access would be inefficient.

5.3 Procedural Integration

The integration of logic with a procedural language works in two directions. Within a procedural program, logical components and inferences are available to obtain more complex functionality. Within a declarative specification, symbols can be interpreted as the result of procedural calls, useful, e.g., for integration with external sources.

In the former direction, logical objects are made available to a procedural language. For that purpose, IDP allows users to write "procedure" components within namespaces that define Lua functions. For example the code `procedure append(left, right){ return left .. right; };` defines a function "append" that

takes two arguments and returns their concatenation (".." is the string concatenation operator in Lua). Each logic object and namespace declared in the global namespace is associated with a corresponding object in the Lua runtime environment, with the same name. Hence, logic object and namespaces are first-class citizens in the Lua stack. Additionally, IDP inferences and functionalities and "procedure" components are available as native Lua functions. For example, the code `procedure onemodel() { return modelexpand(T,S)[1]; };` will pass the variables `T` and `S` to the model expansion inference. If these refer to a theory, respectively structure, in the global namespace, model expansion is executed and the first model is returned[4].

In the latter direction (interpreting logical symbols through procedural calls), in a structure, symbols can be interpreted by procedures as follows. In the context of a structure $\mathcal{I}$, one can specify the interpretation of a predicate symbol `P[T1,..Tn)]` by a Lua boolean function `f` with the same type of arguments as `P = procedure f`. In that case, `P(d1,... dn)` is true (false) iff `f(d_1,..., d_n)` returns "true", respectively "false". For a function symbol `f`, a procedure should, given `d1,..., dn` as input, return the interpretation of `f(d1,..., dn)`, or (for partial functions) `nil` if this term is non-denoting. For example, interpreting a function by string concatenation, an operation that already exists in Lua, can be modelled as:

```
vocabulary V is { func add[string,string->string]; func c[->string] };
theory T over V is { c = add("Hello, ", "world.") };
structure S over V is { add = procedure append };
procedure append(left,right) is { return left .. right };
```

Two restrictions on using a Lua function `f` as interpretation of a symbol s are (i) that `f` is a *pure* function (i.e., the return value should only depend on the input arguments and there should be no side-effects) and (ii) that it results in a consistent interpretation for s that also obeys its type signature.

FO(·) *datamodel for Lua.* An IDP specification organizes its logic components in a hierarchy of namespaces with the namespace *global* at the root. Referencing a concrete element in this hierarchy is done through `n[s]`, with `n` the name of a variable that refers to a namespace N and `s` the name of a logical object, namespace or procedure that belongs to N. In a similar fashion, an access `V[n]`, retrieves the symbol with name `n` from vocabulary `V`, and `S[s]` retrieves the interpretation of the symbol `s` from structure `S`.

For example, calling the procedure `main()` first passes the values of the variables `T` and `S` to the function `onemodel`; next, the symbol c with name `n` is retrieved from the vocabulary `V` and is used to retrieve the interpretation of c from the model found; finally, this interpretation is printed.

```
procedure main() { print(onemodel(T,S)[V[c]]); };
```

[4] If no models exist, `nil` is returned, the Lua equivalent of an empty reference.

As is often the case, many inferences can be customized in some way. For example, how structures should be printed (interpretations of sets or lists of facts), how many models a call to model expansion should (try to) return or how much output should be printed while solving a query, ... In IDP, most of these options are set globally through the `stdoptions` Lua table. For example, `stdoptions.nbmodels=n;` indicates that subsequent model generation tasks should try to find `n` models.

Dynamics. Execution of the procedural program requires semantics for the state of logic components at different points in time. For a component C, we refer to its "version" at time t as C_t, with C_1 that state after parsing. Datamodel operations change the underlying logic components (not just the contents of Lua variables). Dependencies in the declarative component of the language, such as includes, are *dynamic*. E.g., if a vocabulary V includes vocabulary V' and at t, a new symbol P is added to V', then V_{t+1} also contains P. A similar principle holds for structures, theories, ...

Such a behaviour is required to support modularity during execution. For example, consider a robot adding knowledge to his `world` theory that contains basic knowledge about the world around him. One would expect any theory including `world` to automatically refer to the newest knowledge instead to a copy of the old information. If required, dependencies can be decoupled by an explicit `clone` operation, which creates a new logical object with the same contents.

File inclusion. The main method of creating logic objects is through providing a list of files, which are then parsed into the relevant logic objects. To allow distributing knowledge over multiple files, the *require* statement is provided: a file that contains a statement `require file ;` instructs IDP to also load the file `file .` For example, one can construct a file with a structure for each instance of a problem and this file can require to load the file(s) containing the vocabularies, the theories, the task, etc., needed to solve the problem of which the the structure is an instance. It then suffices to load a single file to solve an instance. The system provides built-in packages with general-purpose components. These include predefined vocabularies and theories as well as extra inference methods, and can be included through `require—std name`, with `name` the name of the package. For example, `require—std mx` includes useful procedures related to model expansion.

Interactive shell. Both Lua and FO$(\cdot)^{\text{IDP}}$ are interpreted languages (or, at least, compiled Just-In-Time). This allows IDP to provide an interactive interpreter for the language, similar to interpreters available in, e.g., many Prolog systems, but now by executing Lua code on the fly. The interpreter provides *autocompletion*, suggesting available IDP procedures that match a partial input; the autocompletion also specifies the expected argument types. Files can be loaded dynamically through `loaddependency(file)`.

6 Case Study: Optimization inference

Optimization is the task of given a theory $\mathcal{T}$, a structure $\mathcal{I}$ and a term t, all over the same vocabulary V, finding models of $\mathcal{T}$ that expand (are more precise than) $\mathcal{I}$. This inference captures Herbrand model generation and (bounded) model expansion, both of which were proposed as logic-based methods for constraint solving, respectively in [30] and [52]. In its most general form, we define optimization for typed FO(ID, Agg, PF) as follows. The inference $OPT\,\langle V, \mathcal{T}, \mathcal{I}, t, V_{out}\rangle$ takes as input a theory $\mathcal{T}$, structure $\mathcal{I}$ and term t, all over vocabulary V, and a vocabulary $V_{out} \subseteq V$. Both $\mathcal{T}$ and $\mathcal{I}$ are well-typed and $\mathcal{I}$ interprets all root types (types without supertypes). The inference returns V_{out}-structures $\mathcal{J}$ such that at least one model of $\mathcal{T}$ expanding both $\mathcal{I}$ and $\mathcal{J}$ exists and that expansion is minimal with respect to t.

One approach to optimization, used in IDP, is through *ground-and-solve*: ground the input theory and term over the input structure and afterwards apply a search algorithm that, e.g., uses branch-and-bound to find optimal models.

In the rest of the section, we present how the optimization algorithm in IDP solves an $OPT\,\langle V, \mathcal{T}, \mathcal{I}, c, V_{out}\rangle$ task. The workflow consists of an FO(ID, Agg, PF) grounding algorithm, a search algorithm for the full ground fragment of FO(ID, Agg, PF) and various analysis methods and transformations, that result in a smaller grounding and/or improved search performance. As running example, we take the use-case of generating a valid course schedule; an overview is shown in Figure 2. Note that `timeOf` is not modeled as a function, but as a relation in combination with a functional constraint. This is on purpose. Experimental results are discussed in Section 7.3; alternative approaches in Section 8.

6.1 Preprocessing

Exploiting input-* definitions [43]. As explained below, top-down grounding techniques, as used in IDP, tend to be rather inefficient in case of complex (inductive) definitions [64]. However, to make the input structure more precise, definitions with only two-valued parameters, so-called *input-** definitions [43], can be evaluated in advance, instead of during the main grounding phase.

Evaluation proceeds as follows. Definitions in the theory are split in strongly connected components on the symbolic level, resulting in a theory $\mathcal{T}_{def,0}$. If $\mathcal{T}_{def,0}$ contains a definition Δ that is input-* over $\mathcal{I}_{def,0} = \mathcal{I}$, Δ is removed from $\mathcal{T}_{def,0}$, resulting in $\mathcal{T}_{def,1}$. Definition Δ is then translated into a tabled Prolog program, with the same well-founded model as Δ in $\mathcal{I}_{def,0}$, and evaluated using XSB-Prolog, resulting in the structure $\mathcal{I}_{def,1} \geq_p \mathcal{I}_{def,0}$. This is repeated until no more input-* definitions are present over the current structure (or an inconsistent structure is found). The final structure is denoted as $\mathcal{I}_{def}$, the theory as $\mathcal{T}_{input-*}$.

Next, consider a total definition Δ such that the defined symbols occur neither outside of Δ nor interpreted in $\mathcal{I}_{def}$; this is a so-called *output-** definition [43]. Any structure that satisfies the theory without Δ and does not interpret symbols defined in Δ, can be extended to a model of the theory by evaluating Δ in it. Consequently, output-* definitions do not have to be considered during

Knowledge

```
vocabulary V is {
   type day; type hour; type course;
   type time constructed from { time(day,hour) };
   type session;  courseOf[session -> course];  timeOf[session, time];
   type teacher;  teaches[teacher,course];    teacherOf[session->teacher];
   type student;  takes[student,course];      attends[student,session];
   overlaps[session,session];
};
vocabulary Vout is {
   include V::courseOf;   include V::timeOf;
   include V::attends;    include V::teacherOf;

};
theory T over schedule {
   !s: ?=1 t: timeOf(s,t);
   define { !s1 s2: overlaps(s1,s2) <- s1~=s2 &
                                       ?t: timeOf(s1,t) & timeOf(s2,t); };

   !s1 s2: overlaps(s1,s2) => teacherOf(s1)~=teacherOf(s2);

   !student sess: attends(student,sess) => takes(student, courseOf(sess));
   ...
};
term C over schedule is // number of conflicts
    #({stud s1 s2: overlaps(s1,s2) & attends(stud,s1) & attends(stud,s2)});
```

Inferences

minimize(T, db, C, Vout)

1. Structure consistency check
2. Detect + exploit functions
3. Detect + break symmetries
4. Derive variable bounds
5. Ground with bounds + search
6. Post-process definition

Procedure interface

```
procedure solve(db) {
    m = minimize(T,db,C,Vout);
    conf = value(C,m);
    print("Conflicts = "..conf);
    return m;
};
```

Figure 2. A (simplified) example of a course scheduling problem. For each course, several sessions are organized and each student should attend one session for each course taken. For each session, a time slot, teacher and list of students has to be found that satisfy constraints such as no teacher nor student overlap and a limited number of students in each session. Optimal model expansion is then applied to minimize the number of conflicts for students, through the inference workflow shown on the left.

search and can be evaluated afterwards in a post-processing step. Theory $\mathcal{T}_{def}$ is obtained from $\mathcal{T}_{input-*}$ by removing all output-* definitions.

Checking consistency If the input structure, now interpreting all types, is inconsistent, this should preferably be detected as soon as possible. This can happen for a number of reasons, such as a domain element in T that is not in a supertype T' of T, or a n-ary function f that maps to multiple domain elements for the same input tuple. Each of these is checked by modelling the property as a formula that is true for variable instantiations that violate the property. For example, the formula $T_{ct}(x) \wedge T'_{cf}(x)$ is true for instantiations that violate the property `T subtype of T'` while $\exists y\ y' : y \neq y' \wedge F_{ct}(x_1, \ldots, x_n, y) \wedge F_{ct}(x_1, \ldots, x_n, y)$ (with F the graph of f) is true in structures in which f has multiple images. To each of those formulas, query inference is then applied over structure $\mathcal{I}_{def}$. If any of these queries is not empty, an empty set of models is returned (referred to as `unsat` from now), with the appropriate instantiation as additional (debug) output. For efficiency reasons, part of the consistency check is done on-the-fly when a structure is being manipulated, for example during parsing.

Reducing quantification depth using functional dependencies [16]. The size of the grounding is in general exponential in the nesting depth of quantifiers (as it involves the Cartesian product of the involved domain sizes). One way to reduce the quantification depth, is to detect that symbols can in fact be split in a number of symbols with a smaller arity. Consider, for example, the `timeOf(session,time)` relation (modelled on purpose as a predicate symbol). If one could detect that the second argument *functionally depends* on the first argument (the first uniquely determines the value of the second), then it could be replaced by a new function `timeOf[session−−> time]` instead. With appropriate transformations, the subformula `?t: timeOf(s1,t) & timeOf(s2,t)` is reduced to `timeOf(s1)=timeOf(s2)`, eliminating the quantification over `t`. Detection of functional dependencies is done by deduction: it is checked whether an FO formula that expresses the dependency is entailed by the original theory. For the `timeOf` for example, the functional dependency holds if the theory entails the sentence $\forall s : \exists_{=1} t : timeOf(s, t)$. The phase takes as input the theory $\mathcal{T}_{def}$ and structure $\mathcal{I}_{def}$ and returns $\mathcal{T}_{func}$ and $\mathcal{I}_{func}$, in which entailed functional dependencies have been made explicit and quantifications have been dropped where possible. However, we have to take care to return models in the original vocabulary of the user. As such, additional output-* definitions are created, that define the original symbol in terms of the newly introduced ones. In our example, this would be the definition `!s: timeOf(s,timeOf(s))`.

Exploiting symmetries [26]. It is well-known that symmetries, when present in a problem, can cause a search algorithm to solve the same (sub)problem over and again. For example the "pigeonhole" problem "do n pigeons fit in $n-1$

holes?" is a problem known to be hard for SAT-solvers. Symmetries can be detected and broken on the propositional level [3], but for large problems, even the task of detecting symmetries becomes infeasible. Detecting symmetries on a first-order level is often an easier problem, as much more structure of the problem is explicitly available. For example for an FO specification of the pigeonhole problem, it is almost trivial to detect that all pigeons are interchangeable. The symmetry detection inference in IDP detects a simple, frequently occurring form of symmetries: interchangeable domain elements. Two domain elements are considered interchangeable if they are of the same type and occur only symmetrically in interpreted predicates. Detected symmetries are handled either through the addition, to $\mathcal{T}_{func}$, of sentences that statically break those symmetries, or by passing them directly to the search algorithm (see below); in the latter case, they are exploited using dynamic symmetry breaking methods [26]. The latter has the important advantage that no parts of the search space are excluded a-priori, only parts symmetric to already visited ones.

6.2 Ground-and-solve

Ground [15]. The grounding algorithm visits the resulting theory ($\mathcal{T}_{func}$) in a depth-first, top-down fashion, basically replacing all variables by all their matching instantiations, according to the interpretation of their types (more care is required if only a type's root type is interpreted).

However, such an instantiation might be unnecessary large. Indeed, if the value of a term or formula is already known in the current structure $\mathcal{I}_g = \mathcal{I}_{func}$, given an instantiation of variables quantifier earlier, it should not have been grounded in the first place. The solution is to, again, apply query inference. For example, consider a formula $\forall \overline{x} \in \overline{T} : \varphi$ and an instantiation $\overline{d}$ for the free variables $\overline{y}$ of φ not in $\overline{x}$. In that case, $\overline{x}$ need only be instantiated with tuples $\overline{d}$' for which $\varphi[\overline{y}/\overline{d}, \overline{x}/\overline{d}']$ is not certainly true in $\mathcal{I}_{func}$, which can be solved through a query over the derived vocabulary. For all instantiations of $\overline{x}$ not in the result of that query, we are certain that the subformula is true anyway. In fact, an incomplete (cheaper) query inference can be applied, as any overapproximation will only result in additional grounding. The result is a ground FO($\cdot$) theory.

To ensure models are generated that expand the current structure ($\mathcal{I}_{func}$), the structure is passed to the search algorithm. However, as we are grounding top-down, we can optimize over this: whenever a domain atom or term is generated by instantiating variables, instead of returning the atom or term itself, its interpretation is returned. For example, if true is returned from a disjunct, true can be directly returned upwards, without grounding the rest of the disjunction.

Improving Ground: Approximation and lifted unit propagation [66,65].

The above grounding algorithm exploits information in the structure using the query inference. For example, if a symbol P is interpreted as true on its domain, the sentence $\forall x : P(x)$ will be simply ignored (replaced by $\mathbf{t}$). However, these optimisations are no longer exploited if P is not interpreted in the structure

(even though we could derive exactly the same information from the sentence $\forall x : P(x)$). Observations like this gave rise to the algorithms presented in [66], where instead of using structure $\mathcal{I}_g$ directly, we first compute a more precise structure $\mathcal{I}'_g$ such that all models of T expanding $\mathcal{I}_g$ also expand $\mathcal{I}'_g$. This allows us to detect much earlier that a formula does not need to be grounded and avoids wasting precious time on useless work. As a more precise structure allows us to create a smaller grounding and increase search performance, the ideal structure as input of the grounding step would be the *most precise* structure that captures all models of T that are more precise than $\mathcal{I}_g$. Of course, finding this ideal structure is a task that is even harder than the original problem.

Instead of searching for this ideal structure, IDP's approach is to execute a lifted (approximative) version of the unit propagation that would occur after grounding anyway. The result is stored as a symbolic representation of a structure. Namely, for each symbol P, P_{ct} and P_{cf} are associated with symbolic set expressions S_{ct} and S_{cf}, expressing that P_{ct} (P_{cf}) is interpreted as $S_{ct}^{\mathcal{I}_g}$, respectively $S_{cf}^{\mathcal{I}_g}$. Consider, for example, a theory containing the sentences $\forall x : P(x) \Rightarrow Q(x)$ and $\forall x : P(x) \Rightarrow R(x)$. Symbolic unit propagation then results in, e.g., $\mathcal{I}'_g$ interprets P_{ct} as $\{x \mid P_{ct}(x)\}$ (interpreted in $\mathcal{I}_g$!), P_{cf} as $\{x \mid P_{cf}(x) \vee Q_{cf}(x) \vee R_{cf}(x)\}$ and R_{ct} as $\{x \mid P_{ct}(x)\}$. During the grounding phase, all queries for variable instantiations and the interpretation of atoms and terms are then evaluated relative to this symbolic interpretation, resulting in less instantiations and more precise interpretations. E.g., if in the above Q is interpreted in $\mathcal{I}_g$, then the second sentence will only be instantiated for x's for which that $Q(x)$ holds in $\mathcal{I}_g$ (for $Q_{ct}(x)$).

A symbolic representation of complete lifted unit presentation often consists of complex formulas, which are infeasible to query. However, we can suffice with any approximation of those formulas, as long as the resulting structure is at least as precise as $\mathcal{I}_g$. Consequently, the formulas are simplified to balance the estimated cost of querying against the expected reduction in number of answers.

Search Optimization in IDP relies on the search algorithm MiniSAT(ID) [15] for ground FO($\cdot$) theories. It takes the ground theory as input together with structure $\mathcal{I}_{func}$. The algorithm combines techniques from SAT, CP and ASP through a DPLL(T) architecture [33]. At the core lies the SAT-solver MiniSAT [31], a complete, Boolean search algorithm for propositional clauses. This core is complemented by a range of "propagator" modules that take care of propagation for all other types of constraints in the theory, such as aggregates, definitions and atoms containing functions. Each module is also responsible for explaining its propagations in terms of the current assignment. For a definition Δ, for example, the module checks whether the current assignment satisfies Δ's completion, whether it is stable and, when a total assignment is found, whether it is also well-founded. A similar module takes care of not visiting parts of the search space that are symmetrical to parts already visited (given the symmetries detected in Section 6.1). Optimization is taken care of by yet another module that ensures the search space is visited in a branch-and-bound fashion. Whenever a

model M is found, with value c^M, a constraint $c < c^M$ is added to the ground theory (which raises a conflict, leading to backtracking and additional search).

6.3 Post-processing

As a final, post-processing step, structures returned by the search step are translated back to structures over V. Next, they are merged with S, output-* definitions are evaluated over them and finally, they are projected to V_{out}, resulting in structures S_{out} that are solutions to the original $OPT\,\langle V, \mathcal{T}, \mathcal{I}, c, V_{out}\rangle$ problem.

Note that an output-* definition is only evaluated if it will have an effect on the eventual V_{out}-structure. If symmetry-breaking was applied, additional solutions can be generated by applying the symmetries to the solutions found.

6.4 Practical considerations

To allow for more practical usage, models are generated one at a time, after which search can be continued. For minimization, intermediate (suboptimal) models are also returned one at a time. The latter sequence of suboptimal models is guaranteed to be descending by value of c, so minimization can be aborted at any time and return the best model found (so-called "anytime" property). The system reports if optimality of the last model found is proven; afterwards, search can continue to find different optimal models. It is also allowed that a root type is not interpreted in $\mathcal{I}$, as long as it is defined in $\mathcal{T}$ by an "input-*-definition" [43].

6.5 Scalability and Infinity

The reader might have noticed that structures and groundings can be very large or even infinite (for example, when a predicate or a quantified variable is typed over `int`). Model expansion (and thus, optimization) over infinite structures takes infinite time in general. In IDP, several techniques are applied to address this issue; they often work well in practice.

A first such technique has already been explained in Subsection 6.2. By intelligent reasoning over the entire theory, we can sometimes derive better variable bounds. Suppose for example that a theory contains formulas $\forall x[int] : P(x) \Rightarrow Q(x)$ and $\forall y[int] : P(y) \Rightarrow R(y)$, where Q only ranges over a finite type, say T, but P and R range over int. The first of these sentences guarantees that P will only hold for values such that Q holds, hence P can only hold for values in the finite type T. Thus we know that the second sentence should only be instantiated for y's in T, i.e., by deriving an improved bound for y, the grounding of the second sentence suddenly becomes finite. The first sentence can be handled similarly. We only ground this sentence for y's in T and maintain a symbolic interpretation expressing that P is certainly false outside of T.

Second, the usage of a top-down, depth-first grounding algorithm has the advantage that interpretations can be evaluated *lazily*: (i) instantiations of quantifications can be generated one at a time, and (ii) the interpretation of atoms and

terms needs only to be retrieved for atoms and terms that effectively occur in the grounding. The same advantage applies for symbols that are interpreted by (complex) procedures: the procedures are only executed for effective occurrences.

Third, the search algorithm maintains bounds on the interpretation of function terms, taking constraint in the grounding into account. Consider a constant $c \mapsto int$, which in itself would result in an infinite search space. However, combined with, e.g., a constraint $0 \leq c \leq 10$ in the grounding, the solver reduces $c \mapsto int$ to $c \mapsto [0, 10]$, a finite search space.

A fourth technique currently under development to increase scalability is *Lazy Grounding* [17]. Lazy Grounding is based on the observation that the entire grounding often is not necessary to find solutions to an OPT-problem. Instead, the technique interleaves grounding with search as follows. Initially, it (roughly) splits the input theory into two parts: one part is grounded and the underlying solver performs its standard search algorithm on it. The other part of the theory is *delayed*: the system makes assumptions about it that guarantee that models found by the solver can be extended to models of the entire theory. Whenever these assumptions become violated, i.e., when they are inconsistent with the solver's current interpretation, the splitting of the theory is revised. Consider for example the following theory, which has an infinite grounding, $\forall x[int] : P(x) \Rightarrow \varphi$, with φ a possibly large formula. A smart lazy grounder could delay the grounding of that sentence with the assumption that P is false for all integers. During search, only when an atom $P(d)$ becomes true, is the sentence grounded for $x = d$ and only that ground sentence is added to the search, the remainder still delayed on the assumption that P is false. Whenever the search algorithm finds a structure that satisfies the grounding and does not violate any assumptions, that structure can be straightforwardly extended to a model of the whole theory.

7 In practice

To round up our discussion on the IDP system, we discuss some practical issues.

7.1 Modelling in $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$

It is well-known that there is a significant difference between the approach to develop procedural code and declarative models. In this section, we provide some useful insights on how common declarative modelling tasks can be accomplished in $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$, by developing yet another application related to course administration: a gui-workflow manager that helps students to select the courses they will take. Additional modelling examples can be found in [9] and at `dtai.cs.kuleuven.be/krr/software/idp-examples`. Related resources are mentioned in Section 8, as a number of KR languages, such as ASP, Zinc and ProB, share modelling patterns.

Below, we model part of the flow of a Graphical User Interface (GUI) for an application where students select their courses for the coming years. With

such an application, a user can, for example, simulate a specific flow or even apply model checking to verify temporal properties (the former is available as "progression" inference in IDP, model checking isn't available in IDP yet).

The simplified use-case we consider presents a user with a list of courses he can take, with associated weight (expected amount of work), where the user can select courses he wants to take (and deselect them again) and the system disables selection of courses for which selection would exceed the maximum allowed weight. A form of Linear Time Calculus (LTC) [57] is used as the basis of the specification, which is well-suited to model dynamic systems. We now give a overview of the specification and some important modelling practices.

The first components are a generic LTC vocabulary and theory expressing concepts such as actions and fluents and specifying the frame axioms. Modularity often increases the ease of comprehension of a specification, if combined with readable names. It also increases reusability. The vocabulary creates a symbol for any relevant, reusable concept (*reification*).Types are sets of objects relevant to the domain at hand; properties of those objects are modelled as (non-type) predicates and functions. Types that depend on other types can be modelled through definitions or constructors.

```
namespace ltc is {
vocabulary V is {
   type time subtype of nat;
   type action;                      type fluent;
   pred do[action,time];
   pred holds[fluent,time];          pred initHolds[fluent];
   pred causes[action,fluent];       pred ends[action,fluent];
};
theory T over V is {
   define {
      !f: holds(f,min[time]) <- initHolds(f);
      !f t: holds(f,t+1)     <- causes(f,t);
      !f t: holds(f,t+1)     <- holds(f,t) & ~ends(f,t);
   };
};
};
```

The above components are then used as building blocks for the concrete dynamic system modelled below. It defines the concrete actions (select/deselect) and fluents (selected courses) and course-related concepts. Large arities are an artifact of representing knowledge as a database. $\mathrm{FO}(\cdot)^{\mathrm{IDP}}$ is not limited to such representations and large arities often force the modeller to use deep nesting of quantifiers, reducing performance.

Defined concepts such as the selected courses and whether a course can still be taken are naturally modelled as definitions. On the other hand, sentences are used to model constraints, such as the constraint that only courses can be

selected that are still available. Functional relations are modelled as function symbols, allowing them to be nested, with afore-mentioned advantages.

```
namespace course−selection is {
vocabulary cs−V is {
   include ltc :: V;
   type fluent constructed from {selected(course)};
   type action constructed from {select(course), deselect(course)};
   func weightOf[course−>nat]; func maxweight[−>nat];
   pred takes[course,time];      pred canTake[course,time];
};
theory cs−T over cs−V is {
   include ltc :: T;
   define {
      !c: initHolds(selected(c)) <− false;
      !c t: causes(selected(c),t) <− do(select(c),t);
      !c t: ends(allowed(c),t) <− do(deselect(c),t);
   };
   define {
      !f t: takes(courseOf(f),t) <− holds(selected(f),t).
      !c t: canTake(c,t) <− sum({c':takes(c',t):weightOf(c')})
                            +weightOf(c)=<maxweight;
   };
   !c t: do(select(c),t) => canTake(c,t) & ~takes(c,t);
    ...
};
};
```

7.2 Software

Both IDP[3] and its search algorithm MINISAT(ID) are open-source systems, freely available from `dtai.cs.kuleuven.be/krr/software`. Next to accepting input in the FO$(\cdot)^{\text{IDP}}$, both systems provide interfaces through C++. The search algorithm MiniSAT(ID) also supports input in Clausal Normal Form (CNF), Quantified Boolean Form (QBF, CNF's higher-order relative), ground ASP (LParse format [58]) and FlatZinc [53]. Next to the core systems, several tools are available to support the user in his or her modelling efforts.

IDP DRAW. The system ID^P_{Draw} [40] offers declarative visualisation: given a set of facts over symbols of a vocabulary $\Sigma_{\text{ID}^P_{Draw}}$, symbols interpreted by ID^P_{Draw} related to drawing objects on a screen (such as `polygon(list(coordinate pairs))`), ID^P_{Draw} generates a visual representation.

A common use case is to visualize structures I over some application domain Σ. This is achieved by expressing the visualization as a definition with open

symbols in Σ and defined symbols in $\Sigma_{\mathrm{ID}^P_{Draw}}$. Evaluating that definition over I results in a set of facts which ID^P_{Draw} uses to generate the visualisation.

The system also supports time-dependent behavior, useful in the analysis of dynamic systems, by adding a time parameter to all facts indicating at which point in time they take effect. Graph support has also been added, to simplify layout of sets of objects with complex interactions.

 . The plugin ID^P_E (IDP Integrated Development Environment) is available for Eclipse, a well-known IDE Platform. ID^P_E is still preliminary, but already supports auto-completion, syntax highlighting and calls to IDP itself. Logical operators in ASCII syntax are visualized as their proper respective logical symbols.

7.3 Usage

The main usage of IDP^3 is currently its model expansion inference, which as discussed earlier, is closely related to generating answer sets of logic programs and to solving constraint satisfaction problems. As such, it shares applications with those domains, general examples of which are scheduling, planning, verification and configuration problems. More concretely, some applications have been modelled in [9], demonstrating is applicability as both an approach to replace procedural programming in some cases and as an approach to rapid prototyping due to the small development time. It has been used to analyze security issues in several contexts, with an emphasis on formal approaches that allow intuitive modelling of the involved knowledge [19,39].

The performance of IDP has been demonstrated for example in the ASP competition series, in 2009 [25] (IDP^2), 2011 [13] (IDP^2) and 2013 (IDP^3)[5] and in [9], where it is compared to various existing approaches to specific problems. The performance of the search algorithm MiniSAT(ID) has been demonstrated in [4], where it turned out to be the single-best solver in their MiniZinc portfolio, and in the latest Minizinc challenges [51]. In [9], it is demonstrated that a KR system like IDP can be practically used as a frontend for lower-level solvers (e.g., SAT-solvers), instead of manually encoding problems in SAT or using custom scripting. Experimental results showed that IDP is able to relieve this burden with minimal performance loss and greatly reduced development effort.

IDP^3 is used as a didactic tool in various logic-oriented courses, at KU Leuven, where also the theorem proving turns out to be a useful feature.

8 Related work

Within several domains, research is targeting expressive specification languages and (to a lesser extent) multiple inference techniques within one language. While we do not aim at an extensive survey of related languages (e.g., [49] has a section with such a survey), we do compare with a couple of them. The B language [1],

[5] Detailed results of the 4^{th} ASP competition are not available at the time of writing.

a successor of Z, is a formal specification language developed specifically for the generation of procedural code. It is based on first-order logic and set theory, and supports quantification over sets. Event-B is a variant for the specification of event-based applications. The language Zinc, developed by Marriott et al. [49], is a successor of OPL and intended as a specification language for constraint programming applications (mainly CSP and COP solving). It is based on first-order logic, type theory and constraint programming languages. Within ASP, a number of related languages, originating from logic programs, are being developed, such as clasp [35] and DLV [46]. They support definitional knowledge and default reasoning. Implementations exist for inference techniques like stable model generation (related to model expansion), visualisation, optimization and debugging. A comparison of ASP and FO(ID) can be found in [22]. The language of the Alloy [42] system is basically first-order logic extended with relational algebra operators, but with an object-oriented syntax, making it more natural to express knowledge from application domains centered around agents and their roles, e.g., security analysis.

The following are alternative approaches to model expansion (or to closely related inference tasks). The solver-independent CP language Zinc [49] is grounded to the language MiniZinc [53], supported by a range of search algorithms using various paradigms, as can be seen on `www.minizinc.org/challenge2012/results2012.html`. In the context of CASP, several systems ground to ASP extended with constraint atoms, such as Clingcon [54] and EZ(CSP) [5]. For search, Clingcon combines the ASP solver Clasp [35] with the CSP solver Gecode [36], while EZ(CSP) combines an off-the-shelf ASP solver with an off-the-shelf CLP-Prolog system. The prototype CASP solver Inca [28] searches for answer sets of a ground CASP program by applying Lazy Clause Generation (LCG) for arithmetic and all-different constraints. As opposed to extending the search algorithm, a different approach is to transform a CASP program to a pure ASP program [29], afterwards applying any off-the-shelf ASP solver. CASP languages generally only allow a restricted set of expressions to occur in constraint atoms and impose conditions on where constraint atoms can occur. For example, none of the languages allows general atoms $P(\bar{c})$ with P an uninterpreted predicate symbol. One exception is $\mathcal{AC}(\mathcal{C})$, a language aimed at integrating ASP and Constraint Logic Programming [50]. As shown in [47], the language captures the languages of both Clingcon and EZ(CSP); however, only subsets of the language are implemented [37]. At the moment, IDP works by grounding FO($\cdot$) input to ecnf, a rich extension of cnf and successively calling an ecnf-solver. However, different target languages can be chosen, for example grounding everything to cnf and calling an out-of-the-box sat solver, grounding to asp, etc. By varying the intermediate language, we can make use of different solvers, but lose the advantages of a strongly integrated system, such as interleaving grounding and search.

9 Conclusion

Kowalski's 1974 paper laid the foundations for the field of Logic Programming, by giving the Horn-clause subset of predicate logic a procedural interpretation to use it for programming. More recently, progress in automated reasoning in fields such as SAT and CP made the exploration possible of more pure forms of declarative programming, gradually moving from declarative programming to declarative modelling, in which the user only has to care about the problem specification.

In this paper, we took this development one step further and presented the knowledge-base system IDP, in which knowledge is separated from computation. The knowledge representation language is both natural and extensible, cleanly integrating first-order logic with definitions, aggregates, etc. It provides a range of inference engines and functionalities for tasks encountered often in practice.

IDP is an extensible framework for declarative modelling, in which both language extensions and inference engines can be added with relative ease. It focuses on moving the burden of performance on modelling from the user to the system, demonstrated by the workflow of optimization inference, which is achieved by combining insights from fields such as SAT, CP, LP and ASP.

References

1. J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.
2. Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: An open toolset for modelling and reasoning in Event-B. *STTT*, 12(6):447–466, 2010.
3. F.A. Aloul, K.A. Sakallah, and I.L. Markov. Efficient symmetry breaking for boolean satisfiability. *Computers, IEEE Transactions on*, 55(5):549–558, 2006.
4. Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. Features for building csp portfolio solvers. arXiv:1308.0227 [cs.AI], 2013.
5. Marcello Balduccini. Industrial-size scheduling with asp+cp. In *LPNMR*, pages 284–296, 2011.
6. Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.
7. Michael Bartholomew and Joohyung Lee. Stable models of formulas with intensional functions. In Brewka et al. [10].
8. Hendrik Blockeel, Bart Bogaerts, Maurice Bruynooghe, Broes De Cat, Stef De Pooter, Marc Denecker, Anthony Labarre, Jan Ramon, and Sicco Verwer. Modeling machine learning and data mining problems with FO($\cdot$). In Dovier and Santos Costa [27], pages 14–25.
9. Hendrik Blockeel, Maurice Bruynooghe, Bogaerts Bart, Broes De Cat, Stef De Pooter, Marc Denecker, Anthony Labarre, Jan Ramon, and Sicco Verwer. Predicate logic as a modeling language: Modeling and solving some machine learning and data mining problems with idp3. *CoRR*, abs/1309.6883, 2013.
10. Gerhard Brewka, Thomas Eiter, and Sheila A. McIlraith, editors. *Principles of Knowledge Representation and Reasoning: Proceedings of the Thirteenth International Conference, KR 2012, Rome, Italy, June 10-14, 2012*. AAAI Press, 2012.
11. Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *CACM*, 54(12):92–103, 2011.
12. Wilfried Buchholz, Solomon Feferman, Wolfram Pohlers, and Wilfried Sieg. *Iterated Inductive Definitions and Subsystems of Analysis : Recent Proof-Theoretical Studies*, volume 897 of *Lecture Notes in Mathematics*. Springer, 1981.
13. Francesco Calimeri, Giovambattista Ianni, and Francesco Ricca. The third open answer set programming competition. *CoRR*, abs/1206.3111, 2012.
14. Weidong Chen and David Scott Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, 1996.
15. Broes De Cat, Bart Bogaerts, Jo Devriendt, and Marc Denecker. Model expansion in the presence of function symbols using constraint programming. Accepted for the IEEE International Conference on Tools with Artificial Intelligence (ICTAI) - 2013.
16. Broes De Cat and Maurice Bruynooghe. Detection and exploitation of functional dependencies for model generation. *Theory and Practice of Logic Programming (TPLP)*, 13(4-5):471–485, 2013.
17. Broes De Cat, Marc Denecker, and Peter Stuckey. Lazy model expansion by incremental grounding. In Dovier and Costa [27], pages 201–211.
18. Leonardo Mendonça de Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, 2011.
19. Koen Decroix, Jorn Lapon, Bart De Decker, and Vincent Naessens. A formal approach for inspecting privacy and trust in advanced electronic services. In Jan

Jürjens, Benjamin Livshits, and Riccardo Scandariato, editors, *ESSoS*, volume 7781 of *Lecture Notes in Computer Science*, pages 155–170. Springer, 2013.
20. Marc Denecker. The well-founded semantics is the principle of inductive definition. In Jürgen Dix, Luis Fariñas del Cerro, and Ulrich Furbach, editors, *JELIA*, volume 1489 of *LNCS*, pages 1–16. Springer, 1998.
21. Marc Denecker, Maurice Bruynooghe, and Victor W. Marek. Logic programming revisited: Logic programs as inductive definitions. *ACM Transactions on Computational Logic (TOCL)*, 2(4):623–654, 2001.
22. Marc Denecker, Yulia Lierler, Miroslaw Truszczynski, and Joost Vennekens. A tarskian informal semantics for asp. In *International Conference on Logic Programming (Technical Communications)*, 2012.
23. Marc Denecker and Eugenia Ternovska. A logic of nonmonotone inductive definitions. *ACM Transactions on Computational Logic (TOCL)*, 9(2):14:1–14:52, April 2008.
24. Marc Denecker and Joost Vennekens. Well-founded semantics and the algebraic theory of non-monotone inductive definitions. In Chitta Baral, Gerhard Brewka, and John S. Schlipf, editors, *LPNMR*, volume 4483 of *LNCS*, pages 84–96. Springer, 2007.
25. Marc Denecker, Joost Vennekens, Stephen Bond, Martin Gebser, and Mirosław Truszczyński. The second Answer Set Programming competition. In *LPNMR*, pages 637–654, 2009.
26. Jo Devriendt, Bart Bogaerts, Christopher Mears, Broes De Cat, and Marc Denecker. Symmetry propagation: Improved dynamic symmetry breaking in SAT. In *Proceedings of the 24th IEEE International Conference on Tools with Artificial Intelligence, ICTAI'12*, 2012.
27. Agostino Dovier and Vítor Santos Costa, editors. *Technical Communications of the 28th International Conference on Logic Programming, ICLP 2012, September 4-8, 2012, Budapest, Hungary*, volume 17 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.
28. Christian Drescher and Toby Walsh. Conflict-driven constraint answer set solving with lazy nogood generation. In *AAAI*, 2011.
29. Christian Drescher and Toby Walsh. Translation-based constraint answer set solving. In *IJCAI*, pages 2596–2601, 2011.
30. Deborah East and Mirosław Truszczyński. Predicate-calculus-based logics for modeling and solving search problems. *ACM Transactions on Computational Logic (TOCL)*, 7(1):38–83, 2006.
31. Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.
32. Alan M. Frisch and Peter J. Stuckey. The proper treatment of undefinedness in constraint languages. In IanP. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009*, volume 5732 of *Lecture Notes in Computer Science*, pages 367–382. Springer Berlin Heidelberg, 2009.
33. Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast decision procedures. In *CAV*, pages 175–188, 2004.
34. Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.
35. Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. *Artif. Intell.*, 187:52–89, 2012.

36. Gecode Team. Gecode: Generic constraint development environment, 2013. Available from `http://www.gecode.org`.
37. Michael Gelfond, Veena S. Mellarkod, and Yuanlin Zhang. Systems integrating answer set programming and constraint programming. In Marc Denecker, editor, *LaSh*, pages 145–152, 2008.
38. Todd J. Green, Molham Aref, and Grigoris Karvounarakis. Logicblox, platform and language: A tutorial. In Pablo Barceló and Reinhard Pichler, editors, *Datalog*, volume 7494 of *Lecture Notes in Computer Science*, pages 1–8. Springer, 2012.
39. Thomas Heyman. *A Formal Analysis Technique for Secure Software Architectures*. PhD thesis, Department of Computer Science, KU Leuven, March 2013.
40. IDPDraw: structure visualization `dtai.cs.kuleuven.be/krr/software/visualisation`.
41. Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. Lua – an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.
42. Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM'02)*, 11(2):256–290, 2002.
43. Joachim Jansen, Albert Jorissen, and Gerda Janssens. Compiling input$*$ FO$(\cdot)$ inductive definitions into tabled prolog rules for IDP^3. *Theory and Practice of Logic Programming (TPLP)*, 13(4-5):691–704, 2013.
44. Stephen Cole Kleene. *Introduction to Metamathematics*. Van Nostrand, 1952.
45. Robert A. Kowalski. Predicate logic as programming language. In *IFIP Congress*, pages 569–574, 1974.
46. Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic (TOCL)*, 7(3):499–562, 2006.
47. Yuliya Lierler. On the relation of constraint answer set programming languages and algorithms. In Jörg Hoffmann and Bart Selman, editors, *AAAI*. AAAI Press, 2012.
48. Vladimir Lifschitz. Logic programs with intensional functions. In Brewka et al. [10].
49. Kim Marriott, Nicholas Nethercote, Reza Rafeh, Peter J. Stuckey, Maria Garcia de la Banda, and Mark Wallace. The design of the Zinc modelling language. *Constraints*, 13(3):229–267, 2008.
50. Veena S. Mellarkod, Michael Gelfond, and Yuanlin Zhang. Integrating answer set programming and constraint logic programming. *Annals of Mathematics and Artificial Intelligence*, 53(1-4):251–287, 2008.
51. Minizinc challenge 2012. `www.minizinc.org/challenge2012/results2012.html`.
52. David G. Mitchell and Eugenia Ternovska. A framework for representing and solving NP search problems. In Manuela M. Veloso and Subbarao Kambhampati, editors, *AAAI*, pages 430–435. AAAI Press / The MIT Press, 2005.
53. N. Nethercote, P.J. Stuckey, R. Becket, S. Brand, G.J. Duck, and G. Tack. Minizinc: Towards a standard CP modelling language. In C. Bessiere, editor, *CP'07*, volume 4741 of *LNCS*, pages 529–543. Springer, 2007.
54. Max Ostrowski and Torsten Schaub. Asp modulo csp: The clingcon system. *TPLP*, 12(4-5):485–503, 2012.
55. Stef De Pooter, Johan Wittocx, and Marc Denecker. A prototype of a knowledge-based programming environment. *CoRR*, abs/1108.5667, 2011.
56. Bertrand Russell. On denoting. *Mind*, 14(56):479–493, 1905.
57. Murray Shanahan. *Solving the frame problem - a mathematical investigation of the common sense law of inertia*. MIT Press, 1997.

58. Tommi Syrjänen. Implementation of local grounding for logic programs with stable model semantics. Technical Report B18, Helsinki University of Technology, Finland, 1998.
59. Michael Thielscher. Representing the knowledge of a robot. In Anthony G. Cohn, Fausto Giunchiglia, and Bart Selman, editors, *KR*, pages 109–120. Morgan Kaufmann, 2000.
60. Allen Van Gelder. The alternating fixpoint of logic programs with negation. *Journal of Computer and System Sciences*, 47(1):185–221, 1993.
61. Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.
62. Hanne Vlaeminck, Joost Vennekens, and Marc Denecker. A logical framework for configuration software. In António Porto and Francisco Javier López-Fraguas, editors, *PPDP*, pages 141–148. ACM, 2009.
63. Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. Spass version 3.5. In Renate A. Schmidt, editor, *CADE*, volume 5663 of *Lecture Notes in Computer Science*, pages 140–145. Springer, 2009.
64. Johan Wittocx. *Finite Domain and Symbolic Inference Methods for Extensions of First-Order Logic*. PhD thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium, May 2010.
65. Johan Wittocx, Marc Denecker, and Maurice Bruynooghe. Constraint propagation for first-order logic and inductive definitions. *ACM Trans. Comput. Logic*, 14(3):17:1–17:45, August 2013.
66. Johan Wittocx, Maarten Mariën, and Marc Denecker. Grounding FO and FO(ID) with bounds. *Journal of Artificial Intelligence Research*, 38:223–269, 2010.