IDP\textsuperscript{3}: Combining Symbolic and Ground Reasoning for Model Generation

Broes De Cat\textsuperscript{*}, Joachim Jansen, Gerda Janssens

Department of Computer Science, KU Leuven
broes.decat, joachim.jansen, gerda.janssens@cs.kuleuven.be

Abstract. IDP\textsuperscript{3} is a knowledge-base system, offering a rich, declarative knowledge representation language, a range of inferences and built-in interaction with a procedural language. In this paper, we give an overview of the system and show how multiple inferences are combined to obtain state-of-the-art model generation.

Keywords: declarative modeling, model expansion, program transformation

1 Introduction

IDP\textsuperscript{3} is a knowledge-base system (KBS) \cite{22}, a system\textsuperscript{1} aimed at (1) allowing natural representation of knowledge over an application domain in a formal language (a logic), (2) offering a range of inferences to tackle diverse reasoning tasks, reusing the same knowledge as much as possible, (3) providing a procedural interface to combine multiple inferences to solve complex/compound tasks in the application domain. An example is a KBS storing knowledge about course scheduling at a university. By applying suitable forms of inference, schedules can be generated automatically at the start of the year, hand-made schedules can be verified, existing schedules can be revised, etc., all using the same knowledge.

As knowledge representation language, IDP\textsuperscript{3} offers the logic FO(\cdot)\textsuperscript{IDP} \cite{8}, a logic that extends full First-Order Logic (FO) with aggregate functions (sum, product, etc.), integer arithmetic, partial functions, a type system and inductive definitions. The logic is closely related to the languages developed in the fields of SAT Modulo Theories (SMT) \cite{21} and Answer Set Programming (ASP) \cite{18}, to the Constraint Programming (CP) language Zinc\cite{19} and the Alloy language\cite{15}. A more formal comparison to ASP can be found in \cite{7}.

A number of different forms of inference are already available in the system (and more are part of current research), such as model generation (find (optimal) models of a logic theory), entailment, query evaluation and simulation of dynamic systems. Model expansion for FO(\cdot)\textsuperscript{IDP} is closely related to answer set generation

\textsuperscript{*} Broes De Cat is funded by the Institute for Innovation through Science and Technology Flanders (IWT).

\textsuperscript{1} The name IDP stands for Imperative-Declarative Programming.
and to solving Constraint Satisfaction Problems (CSPs), for example in the systems Gringo[12]-Clasp[11], DLV[17], Comet[20] and Gecode[13].

The procedural interface is available through the language Lua.

In this paper, we present IDP³, the newest version of the IDP system, which turned the model expansion system IDP² into a real KBS. We specifically show how IDP³ works towards one of our main research goals, namely to reduce the dependence of inference performance on the way knowledge is specified. To that end, we zoom in on the model expansion inference in IDP³ (Section 3), improving over IDP² by, among others, definition evaluation using XSB Prolog and search on ground instead of propositional theories (see further).

2 The Knowledge-Base System IDP³

The language FO(·)IDP We focus on the aspects of FO(·) that are relevant for this paper; more details can be found in [6], [1] and modeling examples at dtai.cs.kuleuven.be/krr/software/idp-examples. An FO(·) formula differs from FO formulas in two ways. Firstly, FO(·) is a many-sorted logic: every variable has an associated type and every type an associated domain. Moreover, it is order-sorted: types can be subtypes of others; the types string and int are built-in basic types. Secondly, the standard terms in FO are extended with aggregate terms: functions over a set of domain elements and associated numeric values which map to the sum, product, cardinality, maximum or minimum value of the set. A definition is a set of rules of the form $\forall \bar{x} : p(\bar{x}) \iff \phi(\bar{x})$, where $\phi(\bar{x})$ is an FO(·) formula. They are interpreted according to the well-founded semantics[24], which naturally capture e.g., mathematical definitions.

An FO(·) specification consists of a set of named logical components, such as vocabularies, theories and structures. A vocabulary declares a set of types and typed symbols. A theory consists of sentences and definitions over a vocabulary. A structure specifies factual data over some vocabulary, by providing a (possibly partial) interpretation of the symbols in its vocabulary. As an illustration, an FO(·) specification of NQueens is shown in the upper part of Figure 1.

The model expansion inference Model expansion in IDP³ takes as input a theory $\mathcal{T}$ over a vocabulary $\Sigma$, a (partial) interpretation $\mathcal{I}$ over $\Sigma$ and an “output” vocabulary $\Sigma_{\text{out}} \subseteq \Sigma$ (respectively $\mathcal{T}$, $\mathcal{V}$, $\mathcal{I}$ and $\mathcal{V}_{\text{out}}$ in the running example). The task is to search for interpretations of $\Sigma_{\text{out}}$ such that an extension exists to $\Sigma$ that also extends $\mathcal{I}$ and is a model of $\mathcal{T}$.

The procedural interface IDP³ combines declarative and procedural specifications by making all (named) FO(·)IDP components available as first-class citizens and providing all inferences as method calls in the procedural interface. This integration is available both from the Lua scripting language (as in the example) and from C++; an interactive shell is also available from Lua. An example is shown in Figure 1, where a procedure solve() is defined such that it generates a model and afterwards prints the interpretation of queen in that model.
vocabulary V {
    type pos isa int.  queen(pos, pos).  diag(pos, pos, pos, pos).
}

theory T: V {
    { diag(x, y, x_2, y_2) ← x ≠ x_2 ∧ abs(x - x_2) = abs(y - y_2) .

    ∀x : ∃_1 y : queen(x, y).  ∀y : ∃_1 x : queen(x, y).
    ∀x y x_2 y_2 : queen(x, y) ∧ queen(x_2, y_2) ⇒ ¬diag(x, y, x_2, y_2).
}

structure I: V { pos = {1..10}.  queen cf = {1, 5}. }

Model expansion (theory, structure, V_out)

1. Type inference
2. Extend the input structure
3. Simplify the input theory
4. Ground
5. Search
6. Post-evaluation

Procedure interface

procedure solve() {
    sol = onemodel(T, I, V)
    print(sol[V:queen])
}

3. Approach to model expansion

The “Inferences” part in the running example shows the basic components of
our model expansion workflow. In this section, each of these is discussed in more
detail. If inconsistency is derived at any point during the different stages (e.g.,
when extending I with entailed information), model expansion is aborted.

1. Type Inference [25] An important consideration for natural modelings
    is the ability to drop type specification wherever clear. For example for the
    sentence ∀x : ∃_1 y : queen(x, y), it is clear that both x and y are intended
to be typed as positions (pos). This could have been specified at the quantification
(e.g., ∀x[pos]), however the system is often able to derive the intended type
itself, because the vocabulary is fully typed. The type inference attempts to
derive the intended types as follows\(^2\): for any untyped variable, its type is the
order-minimum type which is a supertype of the types of all argument positions
in which the variable occurs; for every variable, such a supertype is required
to exist. In our example, x and y are typed pos as they both only occur in a
position typed as pos (queen(pos, pos)). The result of the type inference step is
a fully typed theory.

\(^2\) Type inference is a non-equivalence preserving transformation: the sentence ∀x[pos] :
∃_1 y[pos] : queen(x, y) is not logically equivalent with ∀x : ∃_1 y : queen(x, y).
2. Structure Transformations

The ideal structure input to the grounding step would be the most precise structure that captures all models of $T$ that are more precise than $I$. Additionally, we would like the structure to store its interpretation symbolically if possible, as the concrete structure can become very large (e.g., any structure that contains $init$ is in fact infinite). A more precise structure allows us to create a smaller grounding and increase search performance. Symbolic representations are important as we might not need to completely enumerate an interpretation, but e.g., only check whether it contains a tuple of domain elements. We present the different symbolic representations $IDP^3$ supports and how some can be derived automatically from $T$ and $I$.

First, types and unary symbols can be specified as integer ranges, see e.g., $pos$ in the running example. Second, an $n$-ary predicate symbol $P$ can be specified as a Lua procedure that takes tuples $d$ of length $n$ that return true iff $P(d)$ is true (and similarly for function symbols). In the running example, we could have defined $diag$ by a procedural call instead of by a definition. These types of symbolic representation are the responsibility of the modeler.

Third, for the different forms of sentences and definitions, the effects of Unit Propagation can be expressed as $FO(\cdot)$ definitions themselves [26][23]. For a given theory and structure, these $FO(\cdot)$ definitions can then be calculated using XSB [16]. The calculated results can then be used to make the structure more precise.

For example, if a queen is present on $(x,y)$ in $I$, it can be derived that there are no other queens in column $x$ and row $y$, which can be expressed as the definition \{queen$_c$(x,y) $\leftrightarrow \exists y' :$ queen$_ct$(x,y') $\land y \neq y'$\} (ct stands for “certainly true”). Instead of computing this definition in advance, which can be very expensive, a lifted version of BDDs [14] can be used to symbolically execute the propagation a number of times, resulting in an approximation of the full propagation [26]. For each symbol $P$, two such BDDs are derived (one for $P_{ct}$ and one for $P_{cf}$), which are then added to the interpretation of $P$ in $I$.

3. Theory Transformations

Next, the question is how the (now typed) theory can be improved to obtain a smaller grounding. The approaches used are (i) to reduce the type of quantified variables using the (symbolic) structure, and (ii) to exploit functional dependencies to drop quantified variables.

Reducing the quantification size [26] We first note that a quantification over a variable $x$ of type $T$ can also be seen as an instantiation of $x$ with all values in the set \{ $x$ | $T(x)$ \} evaluated in $I$. This can straightforwardly be generalized to a set of variables. Given a formula $\forall \exists \in \{ \exists \ | T_1(x_1) \land \ldots \land T_n(x_n) \} : \phi$, the size of the quantification can be reduced by replacing the set condition with the formula $T_1(x_1) \land \ldots \land T_n(x_n) \land \phi$, where symbols in $\phi$ are replaced by their appropriate ct/cf symbolic interpretation in $I$. This formula is then simplified to balance the cost of its evaluation against the (expected) reduction in number of answers.

For example, for the constraint $\forall x \ y \ x_2 \ y_2 : queen(x, y) \land queen(x_2, y_2) \Rightarrow \neg diag(x, y, x_2, y_2)$, a possible associated set would be: \{ $x \ y \ x_2 \ y_2$ $\neg queen_c_f(x, y) \land \neg queen_c_f(x_2, y_2) \land diag(x, y, x_2, y_2)$ \}. This set can be further improved by replacing $diag$ with its symbolic interpretation, resulting in \{ $x \ y \ x_2 \ y_2$ $\neg queen_c_f(x, y) \land
\neg \text{queen}_{cf}(x_2, y_2) \land x \neq x_2 \land \text{abs}(x - x_2) = \text{abs}(y - y_2)\}. \text{ In the structure in the running example, this would for example eliminate all instantiations where } x = 1 \text{ and } y = 5, \text{ as } \text{queen}(1, 5) \text{ is false in } \mathcal{I}, \text{ and instantiations where } x = x_2.

**Function Detection** [4] It often happens that functional dependencies are present between arguments of the same symbol that are not explicitly modeled as a function. In our running example, the theory entails that \text{queen} is a bijection between rows and columns. We detect implicit functional dependencies by using a theorem prover to prove that the associated constraints are entailed by the theory. If this is the case, the theory is rewritten to explicitly introduce new function symbols, as this can significantly reduce the size of the grounding.

For example, we might replace \text{queen}(pos, pos) with a function \text{f}_q(pos) \mapsto pos, mapping the first argument of \text{queen} to the second. The constraint \(\forall x : \exists_1 y : \text{f}_q(x) = y\) is then dropped as it is trivially true, and the constraint \(\forall x \mathcal{y} x_2 : \text{f}_q(x) \land \text{queen}(x, y) \land \text{queen}(x_2, y_2) \Rightarrow \neg \text{diag}(x, y, x_2, y_2)\) is replaced with \(\forall x x_2 : \neg \text{diag}(\text{f}_q(x), x_2, \text{f}_q(x_2))\). The interpretation of \text{queen} in any model is then taken care of by calculating the definition \{\text{queen}(x, y) \leftarrow \text{f}_q(x) = y\}.

**Definition stratification** [16] Definitions in the theory are stratified whenever possible (if there are no loops between rules in the resulting definitions). A definition for which all open symbols are two-valued in \(\mathcal{I}\) (in that case a unique expansion of the defined symbols exists) is translated into a Prolog program of which the answers are the interpretation of the defined symbols. This program is executed using XSB-Prolog. \(\mathcal{I}\) is extended with the results and the definition is removed from the theory. This is repeated until a fixpoint is reached.

4. **Grounding** [27] The effective grounding phase basically consists of replacing all variables by all their matching instantiations, interleaved (for speed) with bringing the theory into a basic normal form using Tseitin variable introduction. The result is a ground \(\text{FO}(\cdot)\) theory.

The grounding algorithm visits the theory in a depth-first, top-down fashion. One advantage is that the symbolic interpretation can be used lazily: types are generated one tuple at a time and atoms are only evaluated when they would effectively occur in the grounding. The main complexity in the algorithm stems from storing enough information to introduce a minimal number of Tseitin variables, which would otherwise reduce search performance. When returning from visiting a formula, the grounder creates a Tseitin variable representing that formula if needed. This is for example not necessary if a conjunction with a false subformula or a disjunction with a true subformula was visited, in which case “false”, respectively “true” is returned instead. When creating Tseitin variables, the grounder reuses the same Tseitin variable for identical subformulas.

5. **Search** [3] Model expansion in \text{IDP}^3 relies on the state-of-the-art search implementation \text{MiniSAT(ID)} for ground \(\text{FO}(\cdot)\) theories. The latest version of \text{MiniSAT(ID)} uses a search algorithm that combines the SAT-solver \text{MiniSAT}
with propagation algorithms for inductive definitions, uninterpreted functions (encoded lazily into SAT) and finite domain constraints such as aggregates.

6. Post-evaluation [16] In fact, as part of phase 3 (theory transformations), the theory can be further reduced, resulting in an even smaller grounding, by removing all rules of which the defined symbol occurs nowhere else in the theory (until fixpoint). These rules do not have to be considered for search as the interpretation of the symbols they define can be computed in polynomial time, given a model of the remaining theory, by transforming them into a Prolog program and evaluating it using XSB (cfr. definition simplification). An example is the definition added by function detection, where queen occurs nowhere else in the (resulting) theory. In fact, if a symbol does not occur in the output vocabulary, it does not even have to be computed (if no other definition depends on it).

4 Implementation and Applications

Both the KBS IDP$^3$ and the search algorithm MINISAT(ID) are open-source, they are available from dtai.cs.kuleuven.be/krr/software/.

As discussed earlier, model expansion for FO(·)$^{IDP}$ is closely related to answer set generation and solving CSPs. As such, it also shares applications with those domains, examples of which are scheduling, planning, verification and configuration problems. The performance of IDP has been demonstrated for example in the ASP competition series [9] and [2] (IDP$^2$) and the 2013 iteration (IDP$^3$)$^3$.

IDP is also used as a didactic tool in various logic-oriented courses, a.o. at KU Leuven, and an IDE for IDP is available at dtai.cs.kuleuven.be/krr/software/idp-ide.

5 Conclusion and future research

In this paper, we presented the architecture of the IDP$^3$ system, a next-generation of knowledge-base system, together with the workflow of its state-of-the-art model expansion inference. The system is an important step towards our aim of reducing the dependence of inference performance on the way knowledge is modeled. To this end, results and tools from different fields have been combined into one model expansion workflow. Important components are automated (i) derivation of smaller bounds on quantifications, (ii) detection of functional dependencies using theorem proving and subsequent theory simplification, (iii) stratification of definitions and efficient evaluation using tabled Prolog and (iv) a search algorithm combining the state-of-the-art techniques from SAT, ASP and CP. Future research is focusing on all aspects of KBS development, both the language, the inference and the interface between them. Concerning model expansion, we are looking into for example optimizing definitions automatically for efficient evaluating using Prolog and reducing the size of the grounding on-the-fly during search[5].

$^3$ Results of the 4th ASP competition are not yet available at the time of writing.
References

1. Hendrik Blochée, 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(^·^). In Dovier and Santos Costa [10], pages 14–25.
3. Broes De Cat, Bart Bogaerts, Jo Devriendt, and Marc Denecker. Model expansion with finite domain, uninterpreted functions. Accepted for the IEEE International Conference on Tools with Artificial Intelligence (ICTAI) - 2013.
5. Broes De Cat, Marc Denecker, and Peter Stuckey. Lazy model expansion by incremental grounding. In Dovier and Costa [10], pages 201–211.


