A Logical Framework for Configuration Software

Hanne Vlaeminck Joost Vennekens Marc Denecker

Department of Computer Science
Katholieke Universiteit Leuven
Belgium
{hanne.vlaeminck, joost.vennekens, marc.denecker}@cs.kuleuven.be

Abstract

There are many reasons why software can be hard to implement. For important classes of applications, the main source of complexity is the *domain knowledge* that is involved. One such class is that of configuration software, which serves to assist a user in making choices in accordance with certain constraints. For instance, consider an application that helps students compose a study program that complies with all relevant university regulations. The reason why this may be difficult to implement is that these regulations can get quite complicated, making them hard to handle, at least for imperative programming methods. A better approach might be to follow the paradigm of a knowledge base system: explicitly represent the domain knowledge in a declarative way, and implement the behavior of the application by performing various logical inference methods on it. Doing this well, however, requires that a number of different components be got right. Most importantly, we need an expressive and purely declarative knowledge representation language, together with a set of useful inference methods. In this paper, we present a framework for implementing this kind of software, based on a rich extension of first-order logic.

Categories and Subject Descriptors F.3.1 [*LOGICS AND MEANINGS OF PROGRAMS*]: Specifying and Verifying and Reasoning about Programs; F.4.1 [*MATHEMATICAL LOGIC AND FORMAL LANGUAGES*]: Mathematical Logic— Logic and constraint programming; D.2.13 [*SOFTWARE*]: Reusable Software—Domain engineering

General Terms Design, Theory

Keywords Knowledge representation, software engineering, product configuration

1. Introduction

Much research in AI tries to find clever techniques for solving problems that were previously unsolvable. For instance, recent advances in constraint programming allow to solve (large instances of) scheduling problems that were impossible ten years ago. However, if we look at the daily practice of software engineering, we see that such computationally hard problems are vastly outnumbered by more mundane problems. A typical example is that of *configuration software*, where the goal is to help a user fill out a form in accordance with certain constraints. Despite their apparent lack of computational hardness, such applications are not always easy to develop and maintain. An example famous in Belgium country is the *tax-on-web* system, commissioned by the Belgian government to allow citizens to fill in their tax return forms on-line and compute the amount of taxes due. Even though the system is now operational, its development took significantly more time and resources than originally planned. This problem is mainly hard because of the complexity of the *domain knowledge* involved: the first part of the forms alone already comes with an explanatory leaflet of no less than 96 pages; most of this knowledge must somehow find its way into the on-line application. Moreover, since tax laws change every year, this had better be done in such a way that it can easily be adapted later on.

In this paper, we seek to apply AI techniques to simplify the development of such knowledge intensive applications. Our aim here is not to solve previously unsolvable problems, but rather to solve easy problems *better*, that is, to enable applications to be written faster, in less lines of code, and in such a way that they are easier to adapt to changes in the domain knowledge (e.g., a revision of some tax law). We will attempt to do this by means of the *knowledge base* paradigm: instead of counting on the programmer to "compile" his knowledge about the domain and the task to be performed into procedural code, we will represent this knowledge explicitly and automatically derive the desired behavior of the program from it, by means of various forms of logical inference.

For this approach to really work, however, we need to represent the domain knowledge in a purely declarative way. That is, it cannot suffice to just encode solutions to one specific problem in this domain. Instead, we need to be able to represent the domain knowledge *on its own*. Otherwise, we cannot hope to be able to reuse this same representation to solve the many different problems and tasks that a single software application might need to perform. Here, the knowledge base paradigm differs significantly and importantly from other strands of declarative programming. A knowledge base is, by nature, not tied to a specific form of inference; it does not encode a solution to a specific problem, nor is it a program with an operational semantics. It is *only* a representation of domain knowledge, and one should be able to—at least in principle—use it to solve different problems and tasks, requiring different forms of inference, such as deduction, model checking, model generation, update and revision, abduction, learning, etc.

To achieve this, we need a knowledge representation language that is sufficiently expressive, not (only) in the formal sense of the word, but also informally. That is, for each natural language statement that we could find in, e.g., the Belgian tax laws, it should be possible to come up with some formula in the logic that is an "obviously" correct formalization thereof. The need for such expressive languages is apparent throughout the domain of Knowledge Rep-

PPDP'09, September 7–9, 2009, Coimbra, Portugal.

resentation, where we see a general tendency towards increasingly more expressive languages. This is happening e.g. for description logics, (e.g., rule languages such as SWRL), constraint programming languages and answer set programming languages (e.g., aggregates, classical negation, disjunction in the head [Leone et al. 2006]) alike. This motivates us to use the language FO[·], which consists of full first-order logic (FO), together with a number of useful extensions, such as arithmetic, inductive definitions and aggregates [Denecker 2000]. As such, it offers both the fundamental constructs of FO (universal/existential quantification, conjunction, disjunction and negation), as well as the extensions thereof that many practical applications require.

In building a practical system, we of course have to take into account the fundamental trade-off between expressivity of the language and efficiency of its inference algorithms. For instance, given FO's undecidability, there is no hope to use theorem proving for reliable problem solving in full FO. One of the contributions of this paper is to show that all of the inference tasks that arise naturally within the context of configuration software *can* be handled efficiently, even for such an expressive language of FO[·].

Our approach distinguishes itself by the following properties:

- We use a very expressive language to represent the domain knowledge
- We treat this language in an entirely declarative way: while writing the theory, the user does not need to be concerned about how this will later be used. This allows the same knowledge to be used to solve different tasks.
- We have algorithms that are able to efficiently perform different tasks for this expressive language.

These properties distinguish our approach from, e.g [Axling and Haridi 1994], [Soininen and Niemelä 1999], [Subbarayan et al. 2004], where the knowledge representation language is either propositional or only a fragment of FO; or from approaches such as [Knolmayer et al. 2000], where the representation has to be fine-tuned towards the use of a particular inference algorithm such as Forward Chaining. In this paper we show that applications such as product configuration software can be developed in rich extensions of FO, using approximate methods.

We begin by discussing the language FO[·] in Section 2. Section 3 introduces a motivating example. In Section 4, we show that FO[·] is expressive enough to express the relevant domain knowledge. Once we have a theory that represents this domain knowledge, we can proceed in Section 5 with formulating the desired behavior of the application in logical terms. As a proof of concept, we explain in Section 6 how we've implemented the motivating example. Section 7 discusses related work.

2. Preliminaries

We assume familiarity with classical logic. This paper uses the following terminology. An *interpretation* S for a vocabulary Σ consists of a non-empty domain D, a mapping from each function symbol f/n to an n-ary function on D, and a mapping from each predicate symbol P/n to a relation $R \subseteq D^n$. A *three-valued* interpretation $\mathbf{V}$ is the same as a two-valued one, except that it maps each predicate symbol P/n to a function $P^{\mathbf{V}}$ from D^n to the set of truth values $\{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$. Such an $\mathbf{V}$ assigns a truth value to each logical atom $P(\vec{c})$, namely $P^{\mathbf{V}}(c_1^{\mathbf{V}}, \ldots, c_n^{\mathbf{V}})$. The three truth values can be partially ordered according to *precision*:

$$\mathbf{u} \leq_p \mathbf{t} \text{ and } \mathbf{u} \leq_p \mathbf{f}.$$

This order induces a precision order $\leq_p$ on interpretations: $\mathbf{V} \leq_p \mathbf{V}'$ if and only if for each predicate P/n and tuple $\vec{d} \in D^n$, $P^{\mathbf{V}}(\vec{d}) \leq_p P^{\mathbf{V}'}(\vec{d})$.

The language that we use in this paper is that of FO[·], which extends standard FO with a number of useful features, and is also the language used in the IDP system [Mariën et al. 2006].

FO[·] is a multi-sorted logic, in which predicates and functions can be typed. An important feature of FO[·] is that it contains *aggregates*. An aggregate expression can be of one of two forms. The first is

$$F(\{x_1 \ldots x_n \mid \phi(x_1, \ldots, x_n)\})$$

Here F is an aggregation function such as *cardinality*, *sum*, or *average*, and the expression denotes the result of applying this function to the set A of all tuples $(x_1, \ldots, x_n)$ for which $\phi(x_1, \ldots, x_n)$ holds; in case of functions such as *sum*, which operate on numerical values instead of tuples, the projection on the first element x_1 is used, i.e.,

$$sum(\{x_1 \ldots x_n \mid \phi(x_1, \ldots, x_n)\}) = \sum_{(x_1,\ldots,x_n)\in A} x_1$$

The second kind of aggregate expression is of the form:

$$cardinality(\{\phi_1, \ldots, \phi_n\}).$$

This expression denotes the number of true formulas among the ϕ_i. To see when such an expression might be useful, we can consider government regulations concerning things such as scholarship grants; often, there are a number of different criteria that apply, and to be eligible for the grant, a student has to satisfy at least i of the n criteria. If we represent each criterion by some formula $\phi_i(x)$, we can concisely write this down in FO[·] as:

$$\forall x\ cardinality(\{\phi_1(x), \ldots, \phi_n(x)\}) \geq i \Leftrightarrow Eligible(x)$$

A second important feature of FO[·] is that it allows *inductive definitions*, i.e., definitions that recursively define a predicate "in terms of itself". In mathematical texts, such a definition is often written down as a set of cases in which the defined relation holds. FO[·] adopts the same format. For instance, let us consider the following example, taken from the wikipedia page on the topic[1]:

The prime numbers can be defined as consisting of:

- *2, the smallest prime;*
- *each positive integer greater then 1, which is not evenly divisible by any of the primes smaller than itself.*

FO[·] offers a formal syntax for representing such an inductive definition as a set of *definitional rules*:

$$\left\{ \begin{array}{l} Prime(2) \leftarrow \\ \forall x\ Prime(x) \leftarrow\ x > 1 \wedge \neg\exists y : y < x \wedge \\ \qquad\qquad\qquad\qquad Prime(y) \wedge Divisible(x, y) \end{array} \right\}$$

In [Denecker and Vennekens 2007], it was argued that the formal semantics of such an expression in FO[·] coincides with the common-sense meaning of the corresponding inductive definition in mathematics.

Inductive definitions cannot, in general, be expressed in classical logic; for instance, it is well-known that the transitive closure of a relation is not first-order definable (see e.g., [Libkin 2004], Proposition 3.1). Because such definition occur often in practice (e.g., reachability in a graph, being someone's ancestor, and so on), this too is a useful extension of FO.

3. Motivating example

In this section, we analyze one particular piece of configuration software: a system that allows a student to select his study program for the next year. Of course, the university imposes certain re-

[1] `http://wikipedia.org/wiki/Inductive_definition`

strictions on a student's choice, which means that we need *domain knowledge* about what constitutes a valid study program. Typical examples of such restrictions include:

- Certain courses belong to a specific module, and can only be chosen if this module itself is chosen;
- If a module is chosen, then all courses that belong to it must be chosen;
- Compulsory courses have to be chosen;
- Each course is worth a number of credits and the student has to choose 60 credits in total.

This knowledge is *declarative* in nature: it is written down in university regulations, where it exist independently from any piece of software that we might want to develop.

The application now should use this declarative knowledge to perform certain tasks.

- When the user has selected all courses he wants to follow, the application should *check* whether this is valid.
- When the user selects a combination of courses that can never be part of a valid study program, the application should show an appropriate *error message*;
- The application should help the user fill out the form by indicating which other choices are already implied by the selection that the user has made so far.

Conceptually, there is nothing complicated about these tasks. Moreover, given that the number of courses available to a student is never excessively large, there are also no stringent algorithmic requirements. Nevertheless, it would not be trivial to implement this application in a traditional imperative language. The reason for this is that the control flow of the program would need to take into account quite complex dependencies between different selections. For instance, consider the following situation:

1. There is a module *Artificial Intelligence* which contains the courses *Machine Learning* and *Prolog*;
2. There are three optional courses, *Declarative Programming*, *History of Computer Science* and *Embedded Software*, of which the student has to select precisely two;
3. Because of the sizable overlap between *Prolog* and *Declarative Programming*, it is not allowed to take both.

Here are but a few of the propagations that the application should be able to perform:

- Selecting *History of Computer Science* and *Embedded Software* implies that *Declarative Programming* cannot be selected;
- Selecting *History of Computer Science* and choosing not to follow *Declarative Programming* implies that *Embedded Software* has to be selected;
- Selecting *Machine Learning* implies that both *History of Computer Science* and *Embedded Software* must be selected;
- Choosing not to take *Embedded Software* implies that *Machine Learning* cannot be selected;

It should be clear that the full list will be rather long. A naive implementation in an imperative programming language such as C or Java, might include an **if**-statement representing each of these possible propagations. An obvious downside of this approach is that it is hard to make sure that all possible propagations are accounted for. Moreover, such an implementation is especially unsuitable if we also take into account the fact that the regulations will most likely change every year. For instance, suppose that the university decides to allow each student a single exception to the rule that selecting a module means selecting *all* of its courses. From a declarative point-of-view, this is a small change, affecting only one of the many rules that govern the selection procedure. However, in a naively implemented system, it might be quite some work to figure out which of the originally implemented propagations should be removed and which should be added—it might even be easier to start over from scratch.

An experienced programmer would therefore probably end up with a design in which each of the different regulations is represented by some object, which takes care of all the propagations that should be performed because of this regulation. Essentially, such a design will lead to the implementation of an *ad hoc* constraint propagation system, which will take quite a bit of work to get right.

4. Constructing the Knowledge Base

The central idea of our approach is to start from a purely declarative representation of the domain knowledge. This means that we should be able to write down this knowledge in a way which is completely independent from how we might later want to use it. First, this simplifies the knowledge representation task, because we are free to put all considerations about the intended behavior of the application out of our mind for the time being. Second, this also makes the representation reusable: if in the future, we want to implement some other functionality in the same domain, we do not need to change our domain knowledge. For instance, we can imagine that in addition to our application for students, we also want to offer a decision support system for the program director, who has to decide whether a student is allowed certain exceptions to the general rules. One piece of functionality that such a system might offer is to pin-point precisely which of the student's desired choices violate which of the university regulations. Given the appropriate logical inference algorithm, we could completely reuse our representation of the domain knowledge to implement this.

So, in our approach, the first step towards implementing the course selection application is to write down the relevant domain knowledge. As mentioned, we will do this in the language FO[$\cdot$]. It is rather obvious how this would go. In our typed logic, we first need to decide on the types that we will use. Let us choose types *Course* and *Module*, together with the "built-in" type *Integer*. We then have a vocabulary Σ consisting of predicates *Compulsory(Course)*, *NumberOfCredits(Course,Integer)*, *BelongsTo(Course,Module)*, *Selected(Course)* and *Selected(Module)*.

- Certain courses belong to a specific module, and can only be chosen if this module itself is chosen;

$$\forall x\ y : BelongsTo(x, y) \wedge Selected(x) \Rightarrow Selected(y).$$

- If a module is chosen, then all courses in this module must be chosen;

$$\forall x : Module(x) \wedge Selected(x) \Rightarrow \forall y : Belongs(y, x) \Rightarrow Selected(y).$$

- Some courses are compulsory, which means they have to be chosen,

$$\forall x : Compulsory(x) \Rightarrow Selected(x).$$

- Each course is worth a number of credits and the student has to choose 60 credits in total.

$$sum(\{x\ y : NumberOfCredits(y, x) \wedge Selected(y)\}) = 60.$$

This last formula uses one of the constructs for aggregation that are present in FO[$\cdot$]. Without such a construct, one would have to enumerate all possible combinations of courses that result in 60

credits or more. This would yield a large and unintuitive theory, which would have to change considerably each time the university changes the number of credits for even just a single course.

5. Using the Knowledge Base

Once we have a theory T that represents the relevant domain knowledge, we can proceed to formulate the desired behavior of the application in logical terms, relative to T.

5.1 A simple validation system

Let us first assume that the user enters all of his choices manually, without any help from the application. Our job is merely to observe what happens and place this in a logical framework. A first observation is that we need to distinguish two different kinds of predicates in the vocabulary:

- *Compulsory(Course)*, *BelongsTo(Course,Module)* and *NumberOfCredits(Course,Credits)* are all given beforehand and will be completely known to the system;
- *Selected(Course)* and *Selected(Module)* are not known beforehand and need to be filled in by the user.

We will refer to the first kind of predicates as *given* and to the second as *wanted*. Let Γ be the set of given predicates and Ω that of the wanted predicates. Obviously, Γ and Ω partition the set of all predicates in Σ. In logical terms, the fact that the given predicates are known up-front means that the application will have at its disposal some interpretation G for Γ. Note that since all courses, modules and number of credits are known up-front also means that the domain of this interpretation will be finite. The eventual end-state that the application should reach is one in which the user has chosen an interpretation for the wanted predicates such that all the university regulations are satisfied. That is, an interpretation W for Ω such that:

$$W \cup G \models T. \tag{1}$$

Here, $W \cup G$ denotes the interpretation for the vocabulary Σ that interprets all predicates $P \in \Gamma$ by P^G and all predicates $P \in \Omega$ by P^W. One requirement for our system is therefore that it should be able to perform the following task.

Task 1 (model checking)**.** *Given a vocabulary Σ, an FO[$\cdot$] theory T over Σ and a two-valued Σ-interpretation I with a finite domain, model checking is the problem of deciding whether $I \models T$.*

It is well-known that the data complexity of this problem is polynomial time for FO, and this is still the case for FO[$\cdot$][Mariën et al. 2006]. Of course, the theory data complexity is perhaps not an ideal measure, here, since the theory itself may also be subject to change and could grow quite large. However, as long as the nesting depth and formula width (i.e. the maximal number of free variables in a subformula) of additional formulas are bounded, the combined complexity of model checking for FO[$\cdot$] remains in P. In practice, (comprehensible) formulas produced by human experts are bounded in this way.

The user constructs his desired interpretation W only in a gradual way. Initially, the application knows nothing about it, but as it runs it obtains more and more information, checkbox by checkbox. This means the application goes through a sequence $\mathbf{W}_0, \ldots, \mathbf{W}_n$ of *three-valued* interpretations for Ω, that starts out with the least-precise interpretation $\bot_p$ as $\mathbf{W}_0$ (i.e., $\mathbf{W}_0$ assigns **u** to each atom), and grows increasingly more precise as the user fills in the form, i.e.:

$$\bot_p = \mathbf{W}_0 \leq_p \mathbf{W}_1 \leq_p \cdots \leq_p \mathbf{W}_n.$$

If the user has completed his selection, then $\mathbf{W}_n$ should correspond to a *two-valued* interpretation W. How the user precisely goes from $\mathbf{W}_i$ to $\mathbf{W}_{i+1}$ depends on the UI design. The most general case is that, for each course, he has both the option to indicate that he wants to follow it *and* to indicate explicitly that he doesn't. This means he can turn an atom which was **u** into either **t** or **f**. It is also common to see UI's in which the option to explicitly *not* select something is not present. Either way, the UI should also offer a button "done", to allow the user to indicate that he does not want to select any more courses; the last step to reach $\mathbf{W}_n$ is that of switching *all* atoms that are still **u** in $\mathbf{W}_{n-1}$ to **f**.

5.2 Detecting errors

Let us now expand the functionality of this simple system. A first obvious improvement is to be more proactive in our error reporting: instead of waiting until the very end, we should flag errors as soon as they are made. To be more concrete, we know that the user has made a wrong choice, once it becomes impossible to fill out the remaining choices in such a way that the end result will be valid. So, we can report an error if the system reaches an interpretation $\mathbf{W}$ such that there no longer exists a two-valued interpretation W for which:

$$W \geq_p \mathbf{W} \text{ and } G \cup W \models T.$$

It is obvious that this way of reporting errors is sound and complete with respect to criterion (1), i.e., for each sequence $\mathbf{W}_0, \ldots, \mathbf{W}_n$, if we would get an error at some $\mathbf{W}_i$, then the two-valued interpretation $\mathbf{W}_n$ cannot satisfy (1) and, vice versa, if $\mathbf{W}_n$ does not satisfy (1), then we will get an error at some $\mathbf{W}_i$ (albeit that this may only happen for $i = n$).

Task 2 (model expansion)**.** *Let T be an FO[$\cdot$] theory over a vocabulary Σ and let Γ be a subset of the predicates in Σ. Let $\mathbf{V}$ be a three-valued interpretation with a finite domain, such that the projection of $\mathbf{V}$ on Γ is two-valued. A two-valued interpretation S is a solution to the model expansion problem with input $\langle T, \mathbf{V}\rangle$ if $S \models T$ and $S \geq_p \mathbf{V}$.*

Notice that if $\mathbf{V}$ is two-valued, then model expansion simply becomes model checking. It was shown in [Mariën et al. 2006] that model expansion for FO[$\cdot$] is always in NP and, moreover, that there exists input $\langle T, \mathbf{V}\rangle$ for which it is NP-complete. Again, this is speaking just of data-complexity, but the complexity bound continues to hold as long as nesting depth and width are bounded, which we can expect to be the case in practice.

5.3 Auto-completion

A second useful improvement is to help the user by filling out parts of the form for him. Indeed, if a certain new choice is implied by the choices made so far, then we can already fill this in ourselves. Given a three-valued interpretation $\mathbf{W}_i$ and an atom A such that $A^{\mathbf{W}_i} = \mathbf{u}$, we say that the choice for $A = \mathbf{t}$ (or $A = \mathbf{f}$) is *forced* iff it is the case that for each two-valued W such that $W \geq_p \mathbf{W}_i$,

$$\text{if } G \cup W \models T \text{ then } A^W = \mathbf{t} \text{ (or, resp., } A^W = \mathbf{f}).$$

Note that if no such W exists (i.e., if the user has already made some invalid choice), then all atoms are both be forced to be true and forced to be false. To take this possibility into account, we extend the lattice of interpretations with a maximally precise element `inconsistent`, that is more precise than each two-valued interpretation.

Task 3 (model completion)**.** *Given an FO[$\cdot$] theory T over vocabulary Σ and a three-valued interpretation $\mathbf{V}$ - with a finite domain - for Σ, compute the greatest lower bound (w.r.t. $\leq_p$) of all solutions to the model expansion problem $\langle T, \mathbf{V}\rangle$, i.e., construct*

$$\overline{\mathbf{V}} = \cap_{\leq_p}\{S \mid S \geq_p \mathbf{V} \text{ and } S \models T\}.$$

Again, this way of assisting the user is sound and complete w.r.t. (1), i.e., for each W such that $G \cup W \models T$, $W \geq_p \mathbf{W}_i$ iff $W \geq_p \overline{\mathbf{W}_i}$. Every time the user makes a choice, we thus can replace the current three-valued interpretation $\mathbf{W}$ by its completion $\overline{\mathbf{W}}$ and update the form accordingly.

However, this task is computationally quite hard. Let us consider the associated decision problem: given T, $\mathbf{V}$ and some $\mathbf{V}'$, where again $\mathbf{V}$ and $\mathbf{V}'$ have finite domains, decide whether $\mathbf{V}' = \overline{\mathbf{V}}$. Since we can decide whether there exists a solution to the model expansion problem $\langle T, \mathbf{V} \rangle$ by checking whether $\overline{\mathbf{V}} = \texttt{inconsistent}$, there exist theories for which this problem is at least NP-complete. For an upper bound, let us consider the subproblem of deciding whether a single atom is forced to be true/false. This is in co-NP. Hence, we can solve the entire problem in polynomial time, given an oracle to decide the status of each single atom. Therefore, the problem is in level Δ_1^P of the polynomial hierarchy.

Because of these complexity considerations, our actual implementation uses a recently developed *approximation* method [Wittocx et al. 2008], which computes a polynomial time approximation $\tilde{\mathbf{V}}$ of $\overline{\mathbf{V}}$. This algorithm is sound, in the sense that $\tilde{\mathbf{V}} \leq_p \overline{\mathbf{V}}$. The price we pay for tractability is of course that the inequality might be strict.

5.4 Undoing choices

Thus far, we have not yet considered what happens when the user changes his mind halfway through a run of the program. This can manifest itself in two ways: either the user tries to undo one of the choices he made himself, or he wants to deselect one of the courses that the application auto-selected.

The first case is easy. What happens here is that the user turns an atom that was **t** back into **u**. We thus obtain a new state which is *less* precise than the previous one, i.e., the applications goes from some $\mathbf{W}_i$ to a $\mathbf{W}_i' \leq_p \mathbf{W}_i$. Of course, the UI then should display $\overline{\mathbf{W}_i'}$ instead of $\overline{\mathbf{W}_i}$, so undoing a choice also undoes all other choices that were implied by it. Note that we can just use the algorithm of Section 5.3 to compute this, regardless of the order in which the user wants to undo his choices.

The second case is a bit trickier. Here, the user attempts to undo some choice $Selected(C)$ that was made automatically, i.e., one that is **t** in $\overline{\mathbf{W}_i}$ but not in $\mathbf{W}_i$. The problem is of course that $\mathbf{W}_i$ still contains whatever choices it were that implied $Selected(C)$ in the first place. So, switching $Selected(C)$ from **t** to **u** buys the user nothing: he will still either have to undo some of the choices from $\mathbf{W}_i$ that imply $Selected(C)$, or else be forced to reselect $Selected(C)$ in the future. Therefore, we might as well prevent the user from performing such actions, and only allow him to undo those choices that he has made itself. Of course the interface should visually make a difference between choices that are made by the user, which he can undo or change, and the ones we deduced for him, which the user cannot change directly.

There is of course also another alternative and that is to try to automatically undo some of the user's choices so that $Selected(C)$ would no longer be implied. That is, we could look for some $\mathbf{W}' \leq_p \mathbf{W}_i$ which still has some $W \geq_p \mathbf{W}'$ such that $W \cup G \models T$ and $Selected(C)$ is **f** in W. In order not to undo too much of the user's work, this $\mathbf{W}'$ should be as close as possible to $\mathbf{W}_i$, so we would be looking for those $\mathbf{W}'$ such that the set of atoms which are **u** in $\mathbf{W}'$ but **t** in $\mathbf{W}_i$ is minimal. However, there are two main disadvantages to this approach. First, because of this minimality criterion it is computationally hard. Second, in this way we might undo more of the user's choices than he intended, which could cause behavior that is hard to understand and potentially frustrating.

5.5 Explaining errors

In subsection 5.2. we showed how we could see detecting errors as a model expansion task. However, merely detecting an error is of course not very useful. It would greatly benefit the user if we could also explain why his choice is not allowed.

Suppose that the application has reached a three-valued interpretation $\mathbf{V}$, such that no interpretation $S \geq_p \mathbf{V}$ is still a model of T. An explanation of the conflict should consist of two essential components:

- a number of choices made by the user that together cause the conflict.
- a number of sentences of the theory T that explain the conflict.

For instance, imagine that we represented the domain knowledge explained in Section 3 by means of a theory that contains the following sentence:

$$\neg SelectedCourse(PrologProgramming) \wedge SelectedCourse(DeclarativeProgramming). \quad (2)$$

If the user is not aware of this rule and tries to select both courses, a good configuration program would give him the following error report:

> It is not valid to make these choices:
>
> - Prolog Programming, Declarative Programming
>
> because they violate this constrains:
>
> - It is not allowed to follow both Prolog Programming and Declarative Programming.

(Where, of course, the programmer should provide this natural language version of (2).) With this information, the user knows not only how to solve the conflict, but also why there was a conflict in the first place.

Task 4 (explaining inconsistency)**.** *Given an FO[$\cdot$] theory T over vocabulary Σ, and a three-valued interpretation $\mathbf{V}$ for Σ so that it is no longer the case that there exits a $S \geq_p \mathbf{V}$ for which $S \models T$. Find an explanation for the inconsistency, consisting of the following two minimal sets:*

1. *a minimal subset $T' \subseteq T$ such that for each $S \geq_p \mathbf{V}$, $S \not\models T'$,*
2. *a three-valued interpretation*

$$\cap_{\leq_p} \{\mathbf{V}' | \mathbf{V}' \leq_p \mathbf{V} \text{ and for all } S \geq_p \mathbf{V}',\ S \not\models T\}.$$

Again, this task is computationally quit hard. In [Wittocx et al. 2009] a model expansion algorithm is described based on the aforementioned approximation method. This algorithm has the advantage that it can be traced. When the algorithm detects an inconsistency this means that some atom has to be both **t** and **f** at the same time. We can then look at the trace and see which user made selections and which rules were used to infer this.

6. Implementation

We have analyzed in Section 5 the desired behavior of the example application and formulated it in logical terminology. We saw that there are four main logical inference tasks of interest. In this section, we will summarize how we have used these to implement the application.

Recall that we have a theory T in vocabulary Σ, where the predicates of Σ are partitioned into given predicates Γ and wanted predicates $\Omega = \{Selected(Module), Selected(Course)\}$. We are also given an interpretation G for the predicates Γ.

The application has a simple UI, that contains a list of checkboxes for all the courses and modules, as well as a button for final-

izing the selection. The state of the application consists of a three-valued interpretation $\mathbf{V}$ for the vocabulary Σ. The UI visualizes the approximation $\tilde{\mathbf{V}}$ of the completion $\overline{\mathbf{V}}$ of $\mathbf{V}$, in a way which makes a clear distinction between the choices made by the user (i.e., those in $\mathbf{V}$) and those inferred by the application (i.e., those in $\tilde{\mathbf{V}}$ but not in $\mathbf{V}$). By clicking on a checkbox for this second kind of choice, the user can ask for an explanation of why it has been inferred. This is implemented by means of some additional bookkeeping in the algorithm that infers it, similar to the algorithm for explaining errors.

At the start of the application, $\mathbf{V}$ is initialized to interpret Γ by G, the predicate $Selected(Module)$ by the function that maps all modules to **u**, and the predicate $Selected(Course)$ by the function that maps all courses to **u**.

The user can perform three actions, each of which affects the state $\mathbf{V}$ (and, hence, produces the corresponding change to the visualization of $\tilde{\mathbf{V}}$ on the screen):

- He can select a course (or module) c for which is $Selected^{\tilde{\mathbf{V}}}(c) =$ **u**. In this case, we change $\mathbf{V}$ by setting $Selected^{\mathbf{V}}(c) =$ **t**.
- He can unselect a course (or module) c for which $Selected^{\mathbf{V}}(c) =$ **t**. In this case, we change $\mathbf{V}$ by setting $Selected^{\mathbf{V}}(c) =$ **u**.
- He can click the button "done". We then construct a two-valued interpretation F representing his final choices: $Selected^{F}(c) =$ **t** for all c such that $Selected^{\tilde{\mathbf{V}}}(c) =$ **t**; all other $Selected^{F}(c)$ are **f**. We then check whether $F \models T$. If not, we produce an error message and do not change the state of the program; otherwise, we commit F to our student data base and terminate.

Because of our use of the approximation algorithm, all of these tasks run in polynomial time. Moreover, for all of the theories we tried in our experiments, the approximation always managed to achieve optimal precision, apart from certain rare cases involving aggregates.

We have implemented this application by means of a simple Java program that provides a UI, which visualizes the underlies three-valued interpretation and calls the approximation method and the IDP-system [Mariën et al. 2006], which is a model expansion system for FO[·], as described above. This application can be downloaded at the following URL:

`http://www.cs.kuleuven.be/~hanne/demo/`

7. Related work

We are not the first to apply AI research to the problem of developing configuration software. [Soininen and Niemelä 1999] present a system to automatically compute valid configurations without user interaction. This is a slightly different setting to ours, since one of the things that interests us is precisely the demands that are imposed by the interactiveness of the system (responsiveness, error reporting, auto-completion). The language they use is a form of propositional logic programs. Another logic programming approach is that of [Axling and Haridi 1994], which does focus on the interactive aspects. They represent domain knowledge in the Sicstus Object System, using a representation which is specialized towards configuration tasks involving actual physical objects, such as computer components. It seems less suited to represent, for instance, the regulations for tax return forms. [Vanden Bossche et al. 2007] describes an *ontology based* development methodology using OWL (together with, as we understand it, several application-specific extensions), whereas [Subbarayan et al. 2004] describes *binary decision diagram* and *constraint programming* based approaches to developing configuration software. Probably the oldest interactive knowledge base systems are so-called *expert systems* [Nikolopoulus 1997]. These are typically rule based: the domain knowledge is represented as a set of implications, which is then used to reason either by forward chaining (deriving the consequence from its antecedents) or backward chaining (deriving that one of its possible antecedents has to have caused the consequence). Here, the interactions with the user are all governed by the same inference method and, moreover, getting the desired behavior often requires that this method of inference already be taken into account while constructing the theory.

A first thing that all of these approaches have in common when compared to our proposal is that they use a language that is less general than FO[·]. By using FO[·], we hope that our method will be applicable to a significantly larger class of applications. Indeed, this language is not only expressive, but also general, in the sense that it is not geared towards any specific application area. Moreover, it is based on classical logic, the most studied and best understood logic in mathematics and computer science. We could consider, for instance, Answer Set Programming [Gelfond and Lifschitz 1991] as an alternative [Baral 2003], but this is probably going to be less familiar to people than classical logic. Moreover, it might perhaps be less suited to represent, e.g., the regulations for a valid study program, since this domain knowledge does not require any of the epistemic or minimal-model features of ASP, but does contain, e.g, nested quantifiers and equivalences.

A second difference is that it is precisely one of our goals to analyze the use of *different* methods of logical inference to implement the different tasks that arise within a software system, all using the *same* domain knowledge. The above approaches all consider a single form of inference: forward/backward chaining for the expert systems, SLD-resolution for the Prolog systems, model generation for ASP, and so on.

There also exist several more *ad hoc* KR systems that take an approach somewhat similar to ours. For instance, [Balduccini et al. 2006] uses a Java shell on top of an Answer Set Programming system to implement a decision support system for a particular diagnosis-and-repair problem in the context of NASA's space shuttle program. Again, the difference to our work is that we are trying to develop a generally applicable framework, that provides a range of domain-independent inference algorithms. By contrast, the aforementioned system uses a single inference algorithm and implements different tasks by changing, from within the imperative shell, the theory on which they are performed. The way in which these changes are made is particular to the application in question.

In as much as that our method also constructs a formal model of (certain aspects of) the behavior of a software program, it is related to research on *workflow languages* such as [van der Aalst and ter Hofstede 2005]. However, whereas they essentially regard the events/actions that occur in a software program as black boxes, we have a semantic model, which focuses precisely on describing the interactions with the user in terms of the domain knowledge. Therefore, our focus is significantly different. Nevertheless, more semantically oriented workflow languages also exist [Davulcu et al. 1998]. These do not yet go as far, however, as to have a full representation of the domain knowledge, that can be considered separately from the behavior of the system.

8. Conclusions and future work

One important source of complexity in software engineering is that declarative domain knowledge and procedural knowledge about the desired behavior of the system are delicately intertwined throughout a typical software program. In this paper, we presented an approach that is aimed toward separating these two components completely. Our framework consists of the expressive language FO[·], together with a number of different inference tasks for this language. We have shown by means of a detailed example that, using these tools, it is possible to easily and elegantly develop configu-

ration software. All that is required from the software engineer is to, first, write down the domain knowledge in FO[·], and, second, to then define the desired behavior of the system in terms of the different inference tasks that we have discerned.

Similar approaches to developing configuration software, such as [Subbarayan et al. 2004], are essentially propositional in nature. By offering full first-order quantification as well as aggregates, our approach makes the task of writing down the domain knowledge significantly easier. In the field of knowledge representation, there is a general trend towards more expressive languages, acknowledging the practical need for such features. This is seen in Answer Set Programming (e.g., aggregates, classical negation, disjunction in the head, [Leone et al. 2006]), Description Logic (e.g., rule languages such as SWRL) and Constraint Programming. The limiting factor is of course always the efficiency of the inference algorithms. In this paper, we have catalogued all of the reasoning tasks that are needed to implement configuration software, and have shown that all of these can be adequately approximated in polynomial time for the expressive language FO[·].

In future work, we hope to extend our methodology to a larger class of software application. While the three inference tasks we have presented here seem adequate to handle simple configuration software, more complex applications might also require other forms of logical inference, which have yet to be defined and implemented. One particular issue here is that we have currently limited ourselves to finite domains only. There are many applications that require types with potentially infinite domains, such as strings, integers, and so on. Theoretically, our approximation algorithm easily accommodates such types, since, in principle, we do not need to be able to conclude anything about them until the user fills in their values. However, it would of course be more useful to propagate information sooner. For instance, in a calendar system, if a meeting has to be finished by a certain time and the user schedules the start of the meeting at a time which is already past the deadline, we like to be able to flag this error. Such functionality could be achieved by integrating constraint propagation techniques into the approximation algorithm. This is currently the subject of ongoing research.

Acknowledgments

Hanne Vlaeminck is supported by the IWT Flanders and Joost Vennekens is post-doctoral researcher supported by the FWO Flanders.

References

Tomas Axling and Seif Haridi. A tool for developing interactive configuration applications. *Journal of Logic Programming*, 19, 1994.

M. Balduccini, M. Gelfond, and M. Nogueira. Answer set based design of knowledge systems. *Annals of Mathematics and Artificial Intelligence*, 2006.

C. Baral. *Knowledge representation, reasoning and declarative problem solving*. Cambridge university press, 2003.

Hasan Davulcu, Michael Kifer, C. R. Ramakrishnan, and I. V. Ramakrishnan. Logic based modeling and analysis of workflows. In *PODS '98*, 1998.

Marc Denecker. Extending classical logic with inductive definitions. In *NMR*, 2000.

Marc Denecker and Joost Vennekens. Well-founded semantics and the algebraic theory of non-monotone inductive definitions. In C. Baral, G. Brewka, and J. Schlipf, editors, *Ninth International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR*, volume LNAI 4483 of *Lecture Notes in Artificial Intelligence*, pages 84–96. Springer, 2007.

M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New generation computing*, 9:365–385, 1991.

G. Knolmayer, R. Endl, and M. Pfahrer. Modeling processes and workflows by business rules. In *Business Process Management*, 2000.

N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.

L. Libkin. *Elements of finite model theory*. 2004.

M. Mariën, J. Wittocx, and M. Denecker. The IDP framework for declarative problem solving. In *LASH*, 2006.

C. Nikolopoulus. *Expert Systems: Introduction to First and Second Generation and Hybrid Knowledge Based Systems*. 1997.

Timo Soininen and Ilkka Niemelä. Developing a declarative rule language for applications in product configuration. In *PADL*, 1999.

S. Subbarayan, R. M. Jensen, T. Hadzic, H. R. Andersen, H. Hulgaard, and J. Møller. Comparing two implementations of a complete and backtrack-free interactive configurator. In *Proc. of Workshop on CSP Techniques with Immediate Application, CP04*, 2004.

Wil M. P. van der Aalst and Arthur H. M. ter Hofstede. Yawl: yet another workflow language. *Inf. Syst.*, 30(4):245–275, 2005.

M. Vanden Bossche, P. Ross, I. MacLarty, B. Van Nuffelen, and N. Pelov. Ontology driven software engineering for real life applications. In *SWESE*, 2007.

J. Wittocx, H. Vlaeminck, and M. Denecker. Debugging for model expansion. In *ICLP 09*, 2009.

Johan Wittocx, Maarten Mariën, and Marc Denecker. Approximate reasoning in first-order logic theories. In *KR*, pages 103–112, 2008.