Abstract Model Generation in Interactive Consultant

Pierre Carbonnelle[1], Gerda Janssens[2], and Marc Denecker[3]

[1] Katholiek Universiteit Leuven
pierre.carbonnelle@kuleuven.be
[2] Katholiek Universiteit Leuven
gerda.janssens@kuleuven.be
[3] Katholiek Universiteit Leuven
marc.denecker@kuleuven.be

Abstract

In a joint man-machine cognitive system, the human and the machine play specific roles: to make a decision, the human collects relevant information about his environment and goals, and translates them into a language understandable by the machine; the machine makes a recommendation; finally, the human evaluates the recommendation and applies it to the environment as appropriate.

We propose an interactive consultant to whom the user describes his knowledge about the problem in a very expressive language: first order logic with arithmetic. The formulas he enters define the set of constraints to be satisfied. The interactive consultant then proposes concrete solutions that satisfy the formulas, possibly using additional background knowledge, and explains how it derived them. These solutions include the recommendation of the interactive consultant, for evaluation by the human.

Support for this evaluation of recommendation is often overlooked, and yet it is critical for the acceptance of the machine by the human. Evaluating samples of concrete solutions is not an efficient way to build trust in the interactive consultant. So, we explain how to generalize these concrete solutions into abstract models that the user can review in order to assess the correctness of the interactive consultant.

We report on our progress in its implementation.

1 Introduction

As computer systems become more and more intelligent, their interactions with humans become richer. "Augmented Intelligence" describes systems where human and machine work together and learn how to solve a problem by taking advantage of their respective strengths. When this happens, human and machine form a "joint cognitive system"[10].

In a joint man-machine cognitive system, the human and the machine play specific roles: the human collects relevant information about his environment and goals, and translates them into a language understandable by the machine; the machine then makes a recommendation; finally, the human evaluates the recommendation. If the recommendation is appropriate, he applies it to the environment; otherwise, the human changes his translation or the machine with the hope of getting a better recommendation, or loses faith in the machine.

We propose a machine to whom the user describes his knowledge about the problem in a very expressive language: first order logic with arithmetic[7], as taught to many students in secondary schools. We call it an interactive consultant.

Indeed, we humans have the peculiar ability to recognize the similarity between different situations: within a particular class of situations, we characterize each situation by a set of characteristics; we describe each specific situation in that class by assigning different values to these characteristics, by observation or by imagination; we can also represent decisions by assigning values to some other characteristics; finally, using some rules or laws combining these

characteristics, we make the judgment that a particular decision is possible, acceptable or desirable in a given situation.

This set of characteristics and their domain is the vocabulary with which we propose to construct mathematical formulas that embody the laws that must be respected when assigning values to describe a possible (or acceptable, or desirable) decision in a situation. We propose to feed these vocabulary and laws to our interactive consultant, so that he can help us reason with them rigorously, and reach correct decisions in particular situations.

With such a consultant, for example, an engineer may collect the rules describing a configuration problem, and translate them to formulas in first order logic and arithmetic; the machine then proposes concrete solutions that the engineer evaluates. If the solution is inappropriate, he may revise his description of the problem until an acceptable configuration is found.

This feedback loop is essential to the learning process, and is critical for the acceptance of the machine by the human. Unfortunately, the evaluation of solutions by humans is often overlooked when designing machines, and is thus time-consuming. With a traditional design, it's only after repeated use or extensive testing that the user gains confidence in the machine.

To speed up the evaluation process, we propose to generalize these concrete solutions into abstract models that can be easily assessed. Instead of evaluating one solution at a time, the human evaluates families of solutions presented in an easy-to-read tabular form similar to the Decision Modeling Notation[2].

The contributions of this paper are: 1) a description of the capabilities of an interactive consultant; 2) a definition of abstract models and methods to compute them.

This paper is organized as follows: 1) we first present the capabilities of an interactive consultant; 2) we recall the main concepts in first order logic with background theories; 3) we introduce the abstract models and explain how to compute them; 4) we finally describe our implementation and illustrate the benefits of abstract models.

2 Interactive consultant

An interactive consultant is a computerized machine that helps a user come to a decision. It is general purpose, and can be reconfigured to obtain recommendations in engineering (e.g., configuration management), legal reasoning (e.g., tax adviser) or economics (e.g., investment decision).

An interactive consultant has two modes of operation. The first mode is used to enter general knowledge over a class of problems, while the second mode is used to enter knowledge specific to a problem at hand and to obtain recommendations. The knowledge base created in the first mode often contains intensional knowledge that can be re-used from one problem to the next.

In the first mode, the user uses an editor to 1) declare the vocabulary for the class of problems, i.e., the set of symbols (aka, "characteristics") and their domain, and 2) to enter his knowledge about this class of problems, in the form of formulas in first order logic with arithmetic. The formulas are the set of constraints to be satisfied by finding appropriate values for the symbols they contain, within the boundary of their respective domains. A solution to these constraints, i.e., a set of pairs associating variables to their value, constitutes a description of a situation and the associated possible (or acceptable or desirable) recommendation.

While not as expressive as a natural language, first order logic with arithmetic is suitable for directly expressing declarative properties of solutions which cannot be expressed in an imperative computer language. This universal mathematical language is taught to many students

in secondary schools. Fig. 1 and 2 show subsets of the vocabulary and formulas encoding knowledge about triangles and quadrilaterals.

```
1   vocabulary {
2       type type constructed from {triangle, quadrilateral}
3       type subtype constructed from
4           { regular_triangle
5           , right_triangle
6           , rectangle
7           , square
8           , irregular}
9       Type : type
10      Subtype : subtype
11
12      Convex
13      Equilateral
```

Figure 1: Mode 1: Editing the vocabulary (partial view)

```
29  theory {
30      Vertices=3 <=> Sides=3.
31      Vertices=4 <=> Sides=4.
32      Type=triangle      <=> Sides=3.
33      Type=quadrilateral <=> Sides=4.
34
35      // General rules
36      Sides=3 => Convex.
37      (∀n[side]: n≤Sides => Angle(n)<180) <=> Convex.
38      (∀n[side]: n≤Sides => Length(n)=Length(1)) <=> Equilateral.
```

Figure 2: Mode 1: Editing knowledge as formulas (partial view)

In the second mode, the interactive consultant displays an interactive user interface that makes it easier for the user to enter additional knowledge about a particular situation, and to view the recommendation. This additional knowledge is entered in the form of additional constraints on the situation and possible decisions, and is converted to formulas by the interactive consultant. These formulas will also have to be satisfied by the solution of the problem.

Ideally, this user interface is generated automatically from the general knowledge entered in the first mode. Fig. 3 is a typical part of the interactive consultant in the second mode.

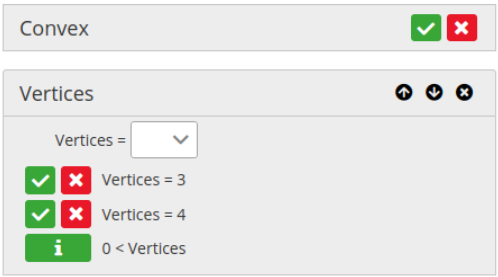

Figure 3: Mode 2 (partial view)

A *decision symbol* is a symbol for which the user needs a value to make his decision in a particular situation. To make a recommendation, the interactive consultant proposes values for the decision symbols.

The interactive consultant elicits information that is relevant for the decision symbols in the particular situation. An information is relevant if it can be used in a proof that a decision symbol has a particular value in a solution. Such a goal-oriented interface is important to reduce clutter when the problem has many data elements, or when the screen is small (e.g., for use on mobile phone). However, the interactive consultant should not be too directive, because users want to keep control of the interactions: the user should be free to choose the order in which he provides information.

When the user enters relevant knowledge, the user interface is updated to reflect their logical consequences, given the general knowledge for this class of problem entered in mode 1. At the minimum, the user interface shows the updated range of values that a symbol may take. It may also display new constraints between symbols, when such constraints become relevant.

The user can ask the interactive consultant to explain how it derives these consequences. It lists the relevant information given by the user concerning the particular situation, as well as the formulas in the knowledge base that lead to the derivation.

We recommend that the user interface do not allow entry of inconsistent information, i.e., information that makes the problem unsolvable. If it does allow it, and if the user does enter inconsistent information, it should then explain why such information makes the problem unsolvable, by showing which formulas cannot be satisfied. The user should be able to undo his last entry.

The user can ask the interactive consultant to find a solution that maximizes a *utility function* symbol, given the formulas. In simpler problems, the user may ask instead for any concrete solution compatible with the general knowledge and the particular situation.

When enough information is entered, the values of the decision symbols become fixed: the user can then see the recommendation for his decision. He can also obtain explanations on how the recommendation was reached.

When insufficient information is entered to fix the decision symbols, the user should be able to see an overview of the classes of possible solutions. For example, if he says that all angles of a polygon are 90 degrees, without any further information, the interactive consultant should be able to tell him that only rectangles and squares are possible, and what are the differences between them. This can be done via abstract model generation, as we propose in this paper.

To respond to the user's input, the interactive consultant performs various reasoning tasks using logic and arithmetic. We now recall key concepts in these fields.

3 Logic with arithmetic

This section gives an overview of logic and arithmetic concepts that we use in this paper. For more details, see [5].

Terms are expressions composed in a language whose vocabulary is a set of *constant*, *function* and *variable* symbols. *Atoms* are expressions composed of a *predicate* symbol and zero or more terms. The number of terms required by each predicate symbol is its *arity*. A *proposition* is a predicate of arity 0. *Ground atoms* are atoms without variables. *Quantifier-free formulas* are expressions composed of ground atoms and *logic connectives*. *Quantified formulas* are expressions composed of a quantifier, variable symbols, and another formula. A quantified formula is *well-formed* if every variable symbol in it is within the scope of the quantifier that declares it. *Well-formed formulas*, or just *formulas*, are expressions composed of ground atoms, well-formed quantified formulas, and logic connectives.

A *model* consists of 1) a non-empty set, called the *universe*, and 2) a *mapping* of symbols in the vocabulary to various elements of the universe, and to functions defined over the universe

(see [5] for details). The model determines a unique *interpretation* of the terms and formulas.

Here, we are interested in models that satisfy a given formula and the *background theories* of equality and arithmetic. These background theories determine the interpretation of the arithmetic symbols such as "=", "<", "+" or "0". A model M *satisfies* a formula F if the interpretation determined by M gives a true value to F and respects the laws of arithmetic. We say that M *is a model of* F.

With this syntactic and semantic apparatus, we can now describe the various reasoning tasks, or *inferences*[7], that the interactive consultant performs:

- the consequences of user's choices are formulas that are true in all models compatible with the formulas representing the knowledge of the situation; the search of consequences is called *model propagation*;
- the explanation on how a consequence is inferred is given by the smallest set of formulas describing the situation that is inconsistent with the negation of the consequence; the search of such a set is called *unsat-core extraction*.
- the solution that maximizes a utility function is obtained by ranking all the models of the formulas representing the knowledge of the situation by the evaluation of the utility function in each of them; this maximization is called *model optimization*.

In this communication, we introduce another inference: *abstract model generation*. We now describe what are abstract models and the method to obtain them.

4 Abstract models

We begin with some new definitions:

Definition 4.1. *The set of* ***atomQs of a vocabulary*** *is the set of ground atoms and well-formed quantified formulas that can be composed with that vocabulary.*

Definition 4.2. *The set of* ***atomQs of a formula F*** *in vocabulary V, noted AQ(F), is obtained by applying these rules recursively:*

** if F is an atomQ of vocabulary V, then AQ(F) = {F};*

** if F is $\neg\phi$, then AQ(F) is AQ(ϕ);;*

** if F is $\phi_1 \bowtie \phi_2$ (where $\bowtie$ is one of the binary logic connectives), then AQ(F) is AQ(ϕ_1) $\cup$ AQ(ϕ_2).*

Definition 4.3.

Given $p : A \to P$, a bijective function from the set A of atomQs of vocabulary V1 to a set P of proposition symbols in vocabulary V2,

the ***propositional skeleton*** *of formula F in V1 with p, noted $PS_p(F)$, is the formula in V2 obtained by applying these rules recursively:*

** if F is an atomQ of V1, then $PS_p(F)$ is p(F);*

** if F is $\neg\phi$, then $PS_p(F)$ is $\neg PS_p(\phi)$;;*

** if F is $\phi_1 \bowtie \phi_2$ (where $\bowtie$ is one of the binary logic connectives), then $PS_p(F)$ is $PS_p(\phi_1) \bowtie PS_p(\phi_2)$.*

For example, the set of atomQs of $p(a) \implies (\exists x : x < a)$ is $\{p(a), \exists x : x < a\}$, and the propositional skeleton can be written as $p1 \implies p2$.

Definition 4.4. *A **LiteralQ** is an atomQ or the negation of an atomQ.*

Definition 4.5. *An **abstract model formula**, AMF, is a conjunction of literalQs: $\bigwedge_i L_i$. An AMF is **canonical** if none of its literalQs can be removed without changing the set of its models.*

Definition 4.6. *A **disjunctive normal form formula with quantifiers**, DNFQ, is either* `False` *or a disjunction of AMFs: $\bigvee_i \bigwedge_j L_{ij}$.*

We can now state the following theorem.

Theorem 4.1. *Any well-formed formula F can be rewritten in an equivalent DNFQ, composed only of atomQs in the set of atomQs of F.*

Proof. To every atomQs of F, we bijectively associate a proposition B_i in a separate vocabulary. We use these propositional symbols to build the propositional skeleton of F. It is well-known that any formula in propositional logic can be rewritten in an equivalent disjunctive normal form using double negation elimination, De Morgan's laws, and the distributive law. By applying the appropriate procedure, we obtain a DNF of the propositional skeleton. By replacing each proposition in it by its associated atomQ, we obtain a DNFQ of F, composed only of atomQs of F. We say it is a DNFQ of F, and it is equivalent to F. □

We can now explain why we call each disjunct of a DNFQ an abstract model formula: given a formula F, each AMF of a DNFQ of F is a formula that describes a (possibly infinite) set of models of F; and, taken together, all the AMFs describe all the possible models of F.

However, as in propositional logic, well-formed formulas have an infinite number of equivalent DNFQs. Hence, the following definition:

Definition 4.7. *A **canonical DNFQ**, denoted CDNFQ, is either* `False` *or $\bigvee_i amf_i$ such that:*
** each amf_i has at least one model, i.e., each AMF is consistent;*
** no model of an amf_i is a model of amf_j, $i \neq j$, i.e., there is no overlap of AMFs;*
** no literalQ of an amf_i can be removed without changing the set of its models, i.e., each AMF is canonical.*

A formula can still have several CDNFQs. For example, the CDNFQs of p could include p and $(p \wedge q) \vee (p \wedge \neg q)$ in a suitable vocabulary.

Theorem 4.2. *Let A be a set of atomQs. Any DNFQ composed of atomQs in A can be rewritten as a CDNFQ composed of atomQs in A.*

Proof. This can be proven easily by defining a syntactic transformation of the DNFQ for each case where a condition is not met: if an amf_i does not have a model, we can safely remove it from the DNFQ; if there is an overlap between amf_i and amf_j, we can replace them by the canonical forms of $amf_i \wedge amf_j$, $amf_i \wedge \neg amf_j$ and $\neg amf_i \wedge amf_j$; if a literalQ can be removed from an AMF without changing the set of its models, we just do so. □

So, one method to find canonical AMFs of a formula F is to apply the syntactic transformations we just described, to obtain first a DNFQ, and then a CDNFQ. Because some of the AMFs obtained by such transformation may not have models consistent with arithmetic, this method may be inefficient. To address this issue, we propose another method, as we now describe.

Theorem 4.3. *A CDNFQ of F is either* `False` *or of the form $amf_1 \vee G$,*
where G is a CDNFQ of $F \wedge \neg amf_1$.

Proof. By definition, a CDNFQ of F is either `False` or of the form $= \bigvee_{i=1}^{n} amf_i$. In the latter case, we can prove that $F \wedge \neg amf_n = \bigvee_{i=1}^{n-1} amf_i$ by using the distributive law, double negation elimination, and the fact that the AMFs do not overlap. By commutativity of the conjunction, this result can be extended to any amf_j of a CDNFQ of F, where $j \neq n$. The theorem ensues. □

This theorem shows that we can build a CDNFQ of F recursively, by finding first an amf_1 of F, then by building a CDNFQ of $F \wedge \neg amf_1$. We can find an AMF of F by finding a concrete model of F and "abstracting it" as we now describe.

Definition 4.8.
*The **abstract model formula for a concrete model M of formula F**, amf^M, is $\bigwedge_{a \in A+} a \wedge \bigwedge_{a \in A-} (\neg a)$ where A_+ (resp. A_-) is the set of atomQs of F whose interpretation in M is true (resp. false).*

Thus, the AMF of a concrete model of F is the conjuction of the atomQs of F that are true in M, and of the negation of the atomQs of F that are false in M.

Theorem 4.4.
If amf^M is an abstract model formula for a concrete model M of formula F, and if G is a DNFQ of $F \wedge \neg amf^M$ then, $amf^M \vee G$ is a DNFQ of F.

Proof. Since the formula proposed in theorem 4.4 is in DNFQ form, we just have to show that it is equivalent to F, i.e., that any model of the proposed formula is a model of F, and vice versa.

If a model I of the proposed formula does not satisfy amf^M, it must satisfy G, i.e, $F \wedge \neg amf^M$, and thus F. If, on the other hand, it satisfies amf^M, it must give the same interpretation to each atomQ of F as model M does, by construction of amf^M. Hence, using the propositional skeleton of F, the interpretation of F in I must be the same as the interpretation of F in M, which is true. Hence I is a model of F.

Similarly, we can show that every model I of F is a model of the proposed formula. This is obviously the case if I satisfies amf^M. If not, we have to show that it satisfies G, i.e. $F \wedge \neg amf^M$, and thus F. This is true per hypothesis. □

Theorem 4.5.
For a formula F composed of n atomQs, any CDNFQ of F has at most 2^n AMFs.

Indeed, there are at most 2^n ways to create non-overlapping AMFs with n atomQs. Hence, any CDNFQ of a finite formula F is finite.

This allows us to propose a terminating method to compute a CDNFQ of formula F, as shown in Table 1. Unlike the syntactic method, it does not generate any AMF inconsistent with arithmetic.

Each `amf` is reduced by the `reduce` auxiliary function (line 8): it removes literals in `amf` that can be removed without changing the set of models, in order to obtain the canonical form. Methods used to find prime implicants (e.g., [8]) and minimum satisfying assignments (e.g., [9]) could be adapted for this purpose. For the sake of brevity, the `reduce` function is not further described.

The `project` function (line 12) handles the special case where the atomQ is undefined, i.e., where the model could be expanded by giving it a true or false value while still satisfying F.

Input:	a formula F
Output:	a CDNFQ of F
Procedure:	1 def CDNFQ(F): 2 solver = Solver() 3 solver.add(F) 4 atomQs = list_of_atomQs_in(F) 5 CDNFQ = [ ] 6 while solver.check() == sat: 7 M = solver.model() 8 amf = reduce(And([project(a_i, M) for a_i in atomQs])) 9 CDNFQ.append(amf) 10 solver.add(Not(amf)) 11 return CDNFQ
Auxiliary function	12 def project(atomQ, model): 13 t = model.eval(atomQ) 14 if t == True: return atomQ 15 elif t == False: return Not(atomQ) 16 else return True # (atom is undefined)

Table 1: Procedure to compute a CDNFQ of formula F (in python syntax).

A typical run of the algorithm for a simple formula `F = (Subtype=regular-triangle ⇔ (Sides=3 ∧ Equilateral))`[1] would find 3 concrete models, from which we create 3 abstract models as shown in Table 2.

Concrete model m {`Subtype,Sides,Equilateral`}	Abstract model formula amf
{`regular-triangle,3,true`}	`Subtype=regular-triangle∧Sides=3∧Equilateral`
{`other,3,false`}	`Subtype≠regular-triangle∧Sides=3∧¬Equilateral`
{`other,4,undefined`}	`Subtype≠regular-triangle∧Sides≠3`

Table 2: Abstract models for `Subtype=regular-triangle ⇔ (Sides=3 ∧ Equilateral)`.

5 Implementation

Our implementation of the interactive consultant is based on prior work[6]. It has been used to design industrial seals and to verify compliance with public tender regulations. It is available online[1].

The user interface for mode 2 of the interactive consultant is automatically generated by extracting the atomQs of the formulas entered in mode 1, and by using them as labels for buttons and fields. The user interface supports most of the functionality described in Section 2, except the goal-oriented approach: the user can not say which symbols are decision symbols; we elicit all information without prioritizing them by relevance.

[1] a regular triangle is a triangle in which all three sides are equal

It uses the Z3 SMT solver[3] for the various inferences. The Z3 solver fully supports linear arithmetic over integer and rationals, but has only partial support for quantifiers and non-linear arithmetic (in the sense that it may or may not be able to solve formulas of that nature).

For demo purposes, it is configured with a knowledge base of triangles and quadrilaterals, and helps a user answer questions about particular polygons, such as "is my polygon convex?". Because of the current partial support of non-linear arithmetic by Z3, it cannot compute surfaces or coordinates of vertices. Other SMT solvers, such as ksmt[4], support non-linear arithmetic, but not quantifiers.

6 Discussion

The abstract models proposed in this paper are useful in the two modes of the interactive consultant.

In the editor mode, the user can select one or more formulas, and click "Check Code": he will get the abstract models for these formulas, in a tabular format.

For example, after selecting `Subtype=regular-triangle` ⇔ `Sides=3` ∧ `Equilateral.`, the user is shown the table in Fig. 4. Each row describes one set of possible solutions, by listing all the literalQs that must be true in each solution. Each cell shows the literalQs containing the symbol of its column heading. The result is identical to the one in Table 2.

Abstract models

Variables:

Subtype	Sides	Equilateral
Not Subtype = regular_triangle	Not Sides = 3	
Not Subtype = regular_triangle	Sides = 3	Not Equilateral
Subtype = regular_triangle	Sides = 3	Equilateral

Figure 4: Abstract models of the correct definition of regular triangle

If instead of using an equivalence in this formula, the user had incorrectly used an "if" (⇐), he would get the abstract models in Fig. 5. As one can see, the 4^{th}, 5^{th} and 7^{th} rows are incorrect solutions. Thus, the user may (in)validate his knowledge base by inspection of abstract models.

In the second mode, the user can enter information on his specific situation, and obtain the list of abstract models, each one representing a set of concrete solutions. Unlike in the first mode, abstraction in the second mode are based on all the formulas in the knowledge base. The number of literalQs can thus be large. To increase the readability of the abstract models, we separately display some literalQs, as in Fig. 6:

- the Given literalQs are those entered by the user in mode 2;
- the Consequences literalQs are direct consequences of the Given literals;
- the Irrelevant literalQs can be made true or false without creating an inconsistency;

Subtype	Sides	Equilateral
Not Subtype = regular_triangle	Not Sides = 3	Equilateral
Not Subtype = regular_triangle	Not Sides = 3	Not Equilateral
Not Subtype = regular_triangle	Sides = 3	Not Equilateral
Subtype = regular_triangle	Not Sides = 3	Not Equilateral
Subtype = regular_triangle	Sides = 3	Not Equilateral
Subtype = regular_triangle	Sides = 3	Equilateral
Subtype = regular_triangle	Not Sides = 3	Equilateral

Figure 5: Abstract models of an incorrect definition of regular triangle

- the Universal literalQs are universally true, irrespective of the user's choice. They are consequences of the general knowledge of the problems.

In the polygon demo, after selecting the atom saying that all angles are right angles, the user can see that there must be 4 vertices. After selecting "Modelexpand / Show all models" in the menu, he obtains the abstract models in Fig. 6. The two rows in the table show that the polygons can now be only squares or rectangles. The second row has a quantified formula to say that, in a square, all sides have the same length as the first side.

Without access to these abstract models, the user would need to see many concrete solutions before coming to that conclusion.

If we had allowed the user to choose a decision symbol (e.g. "Subtype"), we could display this symbol and associated literalQs in the rightmost column of the table, and obtain a decision table.

Abstract models

Given:∀n[side] : n ≤ Sides ⇒ Angle(n) = 90
Consequences: Vertices = 4,Sides = 4,Type = quadrilateral,Angle(1) = 90,Angle(2) = 90,Angle(3) = 90,Angle(4) = 90,Not Vertices = 3,Not Sides = 3,Not Type = triangle,Convex,∀n[side] : n ≤ Sides ⇒ Angle(n) < 180,Not Subtype = regular_triangle,Not ∀n[side] : n ≤ Sides ⇒ Angle(n) = 60,Not Subtype = right_triangle,∃n[side] : n ≤ Sides & Angle(n) = 90,Length(1) = Length(3),Length(2) = Length(4),Length(1) ≤ Length(2) + Length(3),Length(3) ≤ Length(1) + Length(2)
Variables:

Length	Subtype	Equilateral
Length(1) ≠ Length(2)	Subtype = rectangle	Not Equilateral
∀n[side] : Length(n) = Length(1) ⇐ n ≤ Sides	Subtype = square	Equilateral

Irrelevant: Length(2) ≤ Length(3) + Length(1)
Universal: Perimeter = sum{n[side]:n ≤ Sides : Length(n)},sum{n[side]:n ≤ Sides : Angle(n)} = (Sides - 2) * 180,0 < Vertices,0 < Sides,∀x[side] : 0 ≤ Length(x),∀x[side] : 0 ≤ Angle(x)

Figure 6: Polygons with right angles only

7 Conclusion

We have described an interactive consultant capable of providing concrete as well as abstract solutions to its user. This allows the user to easily evaluate the validity of abstract solutions, i.e., of sets of concrete solutions. Validating abstract solutions is more efficient for the user than validating concrete solutions. Our approach thus helps build confidence in the validity of the machine, a critical element in joint man-machine cognitive systems.

References

[1] Abstract configuration tool. https://autoconfigparam.herokuapp.com/. Accessed: 2019-04-29.

[2] Decision model and notation. https://www.omg.org/dmn/. Accessed: 2019-04-29.

[3] The Z3 theorem prover. https://github.com/Z3Prover/z3. Accessed: 2019-04-29.

[4] Franz Brauße, Konstantin Korovin, Margarita Korovina, and Norbert Th Müller. A cdcl-style calculus for solving non-linear constraints. *arXiv preprint arXiv:1905.09227*, 2019.

[5] Sanjit A. Seshia Clark Barrett, Roberto Sebastiani and Cesare Tinelli. *Handbook of satisfiability*, chapter Satisfiability Modulo Theories, pages 737–797. IOS press, 2008.

[6] Ingmar Dasseville, Laurent Janssens, Gerda Janssens, Jan Vanthienen, and Marc Denecker. Combining DMN and the knowledge base paradigm for flexible decision enactment. In *RuleML (Supplement)*, 2016.

[7] Broes De Cat, Bart Bogaerts, Maurice Bruynooghe, Gerda Janssens, and Marc Denecker. Predicate logic as a modeling language: the IDP system. In *Declarative Logic Programming*, pages 279–323. Association for Computing Machinery and Morgan & Claypool, 2018.

[8] David Déharbe, Pascal Fontaine, Daniel Le Berre, and Bertrand Mazure. Computing prime implicants. In *2013 Formal Methods in Computer-Aided Design*, pages 46–52. IEEE, 2013.

[9] Alexey Ignatiev, Alessandro Previti, and Joao Marques-Silva. On finding minimum satisfying assignments. In *International Conference on Principles and Practice of Constraint Programming*, pages 287–297. Springer, 2016.

[10] David D Woods. Cognitive technologies: The design of joint human-machine cognitive systems. *AI magazine*, 6(4):86–86, 1985.