Synthesis of Logic Programs from Object-Oriented Formal Specifications

Ángel Herranz[1] **and Julio Mariño**[1]

1 {aherranz,jmarino}@fi.upm.es
Babel Group, Universidad Politécnica de Madrid

Abstract

Early validation of requirements is crucial for the rigorous development of software. Without it, even the most formal of the methodologies will produce the wrong outcome. One successful approach, popularised by some of the so-called *lightweight formal methods*, consists in generating (finite, small) models of the specifications. Another possibility is to build a running prototype from those specifications. In this paper we show how to obtain executable prototypes from formal specifications written in an object oriented notation by translating them into logic programs. This has some advantages over other lightweight methodologies. For instance, we recover the possibility of dealing with recursive data types as specifications that use them often lack finite models.

1998 ACM Subject Classification F.3.1, D.3.1

Keywords and phrases Formal Methods, Logic Program Synthesis, Object-Oriented, Executable Specifications, Correct-by-Construction

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.95

1 Introduction

Lightweight formal methods [12, 9] have become relatively popular thanks to their success in early validation of requirements, a smooth learning curve and the availability of usable tools. This simplicity is obtained by replacing formal proof – which often demands human intervention – by model checking, but this also implies giving up correctness in favour of a less stringent criterion for models.

Consider, for example, the stepwise specification of queues in Alloy [13]. The specifier might start by just sketching the interface up, like in

```
module myQueue
sig Queue { root: Node }
sig Node  { next: Node }
```

that is, stating that *queues* must have a *root node* and nodes will have a *next* node to follow. The description can be "validated" by fixing a number of Queue and Node individuals and letting a tool like *Alloy Analyzer* [2] model check the specification and show graphically the different instances found. Of course, some of these instances will be inconsistent with the intuition in the specifier's mind – e.g. unreachable nodes or cyclic queues, that can be revealed with very small models. Further constraints, like

```
fact allNodesBelongToOneQueue { all n:Node | one q:Queue | n in q.root.*next }
fact nextNotCyclic {no n:Node | n in n.^next}
```

can be added to the myQueue module in order to supply some of the pieces missing in the original requirements. The first *fact* states that for every node there must be some queue

Technical Communications of the 27th International Conference on Logic Programming (ICLP'11).
Editors: John P. Gallagher, Michael Gelfond; pp. 95–105

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

such that the node lies somewhere in the transitive-reflexive closure of the next relation starting with the root of that node. The second one says that no node can be in the transitive closure of the next relation starting in itself. Model checking the refined specification will generate less instances, thus allowing to explore bigger ones, which will hopefully lead to reveal subtler corners in the requirements.

As said before, this approach is extremely attractive: requirements are refined in a stepwise manner guided by counterexamples found by means of model checking, and the whole process is performed with the help of graphical tools. However, there are also some limitations inherent to this approach. Leaving aside the fact that total correctness of the specification is abandoned in favour of a more relaxed notion of being *not yet falsified by a counterexample*, which can make the whole enterprise unsuitable for safety critical domains, the use of model checking rather than proof based techniques also brings other negative consequences, such as limiting the choice of data types in order to keep models finite, making extremely difficult to model and reason over recursive data types like naturals, lists, trees, etc. (See [13], Ch. 4, Sec. 8.)

A natural alternative to model checking the initial requirements is to produce an executable prototype from them. Using the right language it is possible to obtain recursive code and validation can be guided by *testing*, which might also be automated by tools such as *QuickCheck* [6]. Regarding how to obtain the prototypes, there are several possibilities. One of them is to follow the *correct by construction* slogan and to produce code from the specification, either by means of a transformational approach that often requires human intervention, or by casting the original problem in some *constructive type theory* that will lead directly to an implementation in a calculus thanks to the Curry-Howard isomorphism [21, 5, 4, 20].

Another possibility is to use logic programming. In this case, executable specifications are obtained *free of charge*, as resolution or narrowing will deal with the existential variables involved in any implicit (i.e. non-constructive) specification. Readers familiar with logic programming will remember the typical examples – obtaining subtraction from addition for free, sorting algorithms from sorting test, etc. – and those familiar with logic program transformation techniques will also recognise that these can be used to turn those naive implementations into decent prototypes. However, when it comes to practical usage, none of these formalisms can compete with the lightweight methods above, due to the great distances separating them from the notations used for modelling object oriented software.

This paper studies the synthesis of logic programs from specifications written in an object oriented notation. The specification language, Clay, is being designed around two driving ideas. First, the language must be small but make room for the basic constructs in object oriented programming. Second, specifications must admit at least one translation into executable prototypes to allow the specifier to interactively validate her own specifications. We contribute, on one hand, a *static* theory of types and inheritance that somehow copes with bridging the aforementioned gap between object orientation and logic programming, and, on the other, a *dynamic* part that deals with search in the presence of equality and inheritance (Section 3). To support our contributions we review the examples of the prototypes automatically generated by our tool (Section 4).

2 Object Oriented Specifications in Clay

Clay is a stateless object oriented formal notation, a class-based language with a nominal type system. Classes are defined as algebraic types in the form of case classes: complete and

```
class BSTInt {
  state Empty { }
  state Node { data : Int , left : BSTInt, right : BSTInt }
  modifier insert (x : Int) {
    post { self : Empty ∧
           result = BSTInt.mkNode(x,BSTInt.mkEmpty,BSTInt.mkEmpty)
         ∨ self : Node ∧
           ( x < self.data : True ∧
             result = BSTInt.mkNode(self.data,self. left . insert (x), self . right )
           ∨ x = self.data ∧ result = self
           ∨ x < self.data : False ∧
             result = BSTInt.mkNode(self.data,self. left , self . right . insert (x))) }
  }
  modifier remove (x : Int) {
    post { result .contains(x) : False ∧ result . insert (x)=self}
  }
  observer contains (x : Int) : Bool {
    post { self : Empty ∧ result : False
         ∨ self : Node ∧ x = self.data ∧ result : True
         ∨ self : Node ∧ x < self.data : True ∧ result = self. left .contains(x)
         ∨ self : Node ∧ x < self.data : False ∧ result = self. right .contains(x) }
  }
}
```

Figure 1 Binary search trees in Clay

disjoint subclasses of the defining class. Classes can be extended by subclassing. Methods are specified with pre and postconditions, first order formulae involving self (the recipient), parameters and result (the resulting object). Atomic formulae are equalities (=) and class membership (:).

An *interlingua* [3] declarative semantics for Clay is provided by translation into first-order logic. Clay tools generate an axiomatisation in Prover9/Mace4 [22] syntax. Then, early detection of inconsistencies is achieved by the combination of automatic theorem proving (Prover9) and model checking (Mace4) of the first order logic theories that reflects the structure of Clay specifications. For the purposes of this paper, the move to logic program synthesis requires, on the *front-end* of the tools, to take some simplifying decisions in order to keep the resulting theory tractable and readable: no multiple inheritance, no overloading (just method refinement) and no parametric polymorphism.

Figures 1 and 2 contain examples of Clay specifications that will guide the whole paper.

2.1 Modelling Data

Let us start with a specification of binary search trees of integers (Figure 1).[1] Instances of a class are the disjoint and complete sum of the instances of its case classes (indicated with keyword **state** due to their similarity to the design pattern *State* [8]): if t is an instance of BSTInt (t : BSTInt) then it is an instance of Empty or, exclusively, of Node. The following Clay formula expresses it formally:

$$\forall\ t\ :\ \mathsf{BSTInt}\ ((t\ :\ \mathsf{Empty} \vee t : \mathsf{Node}) \wedge t : \mathsf{Empty} \Leftrightarrow \neg\ t : \mathsf{Node})$$

[1] Clay allows parametric polymorphism but we have not used this feature for the sake of conciseness.

```
class Cell {
  state CellCase { contents : Int }
  constructor mkCell {
    post { result.contents = 0 }
  }
  observer get : Int {
    post { result = self.contents }
  }
  modifier set (v : Int) {
    post { result.contents = v }
  }
}
```

```
class ReCell <: Cell {
  state ReCellCase { backup : Int }
  constructor mkReCell {
    post { result = Cell.mkCell ∧
           result.backup = result.contents }
  }
  modifier set (v : Int) {
    post { result.backup = self.contents }
  }
  modifier restore {
    post { result.contents = self.backup ∧
           result.backup = self.backup }
  }
}
```

Figure 2 Inheritance in Clay

The case classes Empty and Node introduce the constructor methods mkEmpty and mkNode. Both are messages that can be sent to the object BSTInt (classes are objects in Clay): BSTInt.mkEmpty creates an instance of the case class Empty and BSTInt.mkNode(42,BSTInt.mkEmpty,BSTInt.mkEmpty) creates an instance of Node.

Composition Composition is represented by fields defined in a case class. Those fields are methods that project the encapsulated information of its case. In our example the result of the expression BSTInt.mkNode(42,l,r).data is 42.

Inheritance Classes can be extended with subclasses that inherit all the properties of the superclass. In Figure 2, class ReCell extends Cell and therefore its instances obey the property: ∀ c : ReCell (c : CellCase).

Inheritance induces a subtype relation (<:) with all its expected laws: reflexivity, transitivity and subsumption. The most important aspect of this relation is that subclasses cannot invalidate by overriding any property specified in a superclass, otherwise the whole specification is considered inconsistent.

This approach is essential when we are specifying in the large: the specifier needs to reason locally to a class and a subclass cannot show a behaviour that forces the specifier to take into account all the subclasses. The approach adds another advantage: specifications can be much more concise since it is not needed to state already stated properties in superclasses. The main drawback is certain loss of flexibility but, in our view, the decision pays back.

The Cell and ReCell classes in Figure 2 are brought from [1]. Instances of Cell are storage-cell objects encapsulating a natural number that can be changed (set) and read (get). The extension of Cell with a restore option yields ReCell. We can observe the conciseness of the overriding of set in ReCell since properties of Cell are inherited.

2.2 Modelling Behaviour

Methods are specified with first order formulae that relate the receiver of the message (self) and the message's parameters with the answer to the message (result). Primitive predicates include equality and class membership.

Class membership is mainly used to do pattern matching. In the specification of method insert we can see how antecedents of implications distinguish between empty and nonempty trees (self : Empty and self : Node).

Equality is particularly interesting in Clay. The predicate is implicitly indexed by the minimum subtype of the compared instances in the context in which the formula appears. The rationale behind this decision has to do with reasoning locally, a more *dynamic* equality would lead unexpected results in the specifier context. In the insert example, the minimum type of result and self in the first disjunct of the post is BSTInt. The semantics establishes that no properties of self other than those *reachable* from BSTInt are enforced in result. In the Cell/ReCell example, formula Cell.mkCellCase(42) = ReCell.mkReCell.set(13).set(42) holds, as only the state relevant in class Cell – the smallest common subtype – is considered.

Keywords **modifier** and **observer** are merely type informative (the result of a modifier is an instance of the class being specified) and has no influence in the semantics of the methods.

In the binary trees example, the specification of insert and contains are, so to say, *explicit* as they describe recursively all the possible situations. The specification of remove, however, is *implicit*: the result is specified by means of a condition that the result must meet with no additional clues.

2.3 Interacting with Clay

The prototype generated by our synthesiser supports interacting with Clay specifications by asking it to reduce a Clay object expression to a normal representation. We describe now some use cases and in Section 4 we will check the actual performance of the synthesised prototype with those use cases.

Inheritance Classes Cell and ReCell will be our first guiding example. We will interact with Clay to check that the compiler is enabling the specifier to write concise specifications with safe inheritance, and we will see the answers to expressions like ReCell.mkReCell.set(0).set(1).get and ReCell.mkReCell.set(0).set(1).restore.get.

Recursive Specifications More interesting examples are the recursive definitions of methods insert and contains of binary search trees in Figure 1. We will interact with the synthesised prototype to check that recursive definitions can be executed.

Implicit Specifications Our last guiding example will be the implicit specification of method remove in the class BSTInt in Figure 1. We will execute some examples sending the message remove to some binary search trees.

Requirements Validation The interaction with Clay should help the specifiers to gain confidence in their specifications. We will detect an error in our previous specifications.

3 Translating Clay Specifications into Logic Programs

The distance between Clay and Prolog is big enough to make the the translation far from trivial and difficult to follow. Its full formalisation can be found in [10]. This section discusses the intuitions behind the main decisions.

Given a Clay specification we will synthesise facts that represent its abstract syntax tree: classes, inheritance, case classes, fields, and pre- and post-conditions of methods. Figure 3 describes the meaning of the target predicates.

The heart of our translator is a common theory for all specifications: *the Clay theory*. The most important predicates of this axiomatisation are (`instanceof/2`, `reduce/2`, and `eq/3`), definitions that rely on the facts translated from the source specifications (Figure 3). Their meaning is:

- Predicate `instanceof(`*NF*`,`*A*`)` is a generator of instances *NF* of a class *A*. *NF* is a *normal form* of an instance of *A*. These normal forms are flexible representation of instances as incomplete data structures and will be presented in Sections 3.1 and 3.2.

`class(`C`)`	C is a Clay's class identifier.
`inherits(`A`,[`B`])`	B is the superclass of A
`cases(`C`,`Cs`)`	Cs is the list with case classes of class C
`fields(`C`,`Fs`)`	Fs is the association list with the field names and field types of case class C
`msgtype(`C`,`M`)`	M is the message identifier of a method defined or overridden in class C
`pre(`C`,`S`,`M`,`As`)`	Precondition for sending message M with arguments As to an instance S of class C
`post(`C`,`S`,`M`,`As`,`R`)`	Postcondition that establishes that R is the resulting instance of sending message M with arguments As to instance S of class C

Figure 3 Representing Clay in Prolog.

- Predicate `eq(`A`,`NF_1`,`NF_2`)`, Clay's equality, decides if the representations (NF_1 and NF_2) of two instances are indistinguishable in class A.
- Finally, predicate `reduce(`E`,`NF`)` reduces any Clay object expression E to its normal form NF. Predicates `eq` and `reduce` will be presented in Sections 3.2 and 3.3.

3.1 Representing Clay Instances in Prolog

We have mentioned that the predicate `reduce/2` reduces a Clay object expression to a normal form. Clay object expressions have a straightforward representation in Prolog:

- A class expression $ci{<}C_1, \ldots, C_n{>}$ is represented by the Prolog term
$$(ci{<}C_1, \ldots, C_n{>})^{\#} = ci^{\#}(C_1^{\#}, \ldots, C_n^{\#}).$$
- A class identifier ci is represented by a valid Prolog constant $ci^{\#}$ by quoting its lexeme.
- A class variable cv (an object variable ov) is represented by a valid Prolog variable $cv^{\#}$ ($ov^{\#}$) by prefixing its name with "_": $\_cv$ ($\_ov$).
- A send expression $o.mi(o_1, \ldots, o_n)$ is represented by the Prolog term
$$(o.mi(o_1, \ldots, o_n))^{\#} = o^{\#}{<}\text{-}mi^{\#}(o_1^{\#}, \ldots, o_n^{\#}).$$
- A message identifier mi is represented by a valid Prolog constant $mi^{\#}$ by using its lexeme.

To describe how the generated prototype represents the instances of our language in normal form we will use the example of restorable cells (instances of ReCell). We need to capture all the information of known superclasses (Cell) and to capture all the information about the specific case class (ReCell).

With no multiple inheritance, a sorted linear structure can represent the classes of an instance. Therefore, we can use a list where each element contains the part of the representation for a given class of the instance: `(`C`,`S`,`F`)` where C is the class, S is the particular case class, and F is an association list from field names to the representation of their instances. Let us show the representation of Cell.mkCell:

```
[('Cell','CellCase',[(contents,[('Int','Int',[42])])])]
```

The list contains one element since the object is an instance of just one class (Cell).

Under subtyping, during a deduction process where a cell with 42 is expected an instance of ReCell could appear. If we follow our rules, the representation of ReCell.mkReCell.set(42) would be:

```
[('Cell','CellCase',[(contents,[('Int','Int',[42])])]),
 ('ReCell','ReCellCase',[(backup,[('Int','Int',[0])])])]
```

The representation of the cell with 42 and the instance of ReCell are partially the same but the latter does not *fit* in the former. This is something that we would expect to happen since both instances represent the same information with respect to the properties of Cell.

We propose to make room for yet unknown information of subclasses and to use an incomplete data structure where the incomplete part represents the *room* for the information of the potential subclasses. The representation of Cell.mkCell would be

```
[('Cell','CellCase',[(contents,[('Int','Int',[42])|_])])|_]
```

and for the instance of ReCell we would have the following representation:

```
[('Cell','CellCase',[(contents,[('Int','Int',[42])|_])]),
 ('ReCell','ReCellCase',[(backup,[('Int','Int',[0])|_])])|_]
```

Apart from carrying all the information needed by methods specified in the superclasses, our normal form has the following properties:

- Information about case classes allows us to reflect the disjoint sum (case classes) of products (fields).
- The incomplete part might be instantiated with data of an instance of a subclass (like the backup of ReCell) during the deduction process. The most interesting benefit is that the instantiation can be implemented with the unification of our logic language engine. The example above shows how the instance of ReCell fits, by unification, in the cell with 42.

Predefined Integers

The predefined class `Int` encapsulates integers that get translated into Prolog integers managed via finite domain constraints. This illustrates another technique that can be applied in the translation when the target language has declarative extensions. Previous versions of the same specification used a Peano representation for naturals (predefined class `Nat`) as a way of obtaining a complete theory for numbers. The experiments in Section 4 show drastic gains over our previous implementation presented in [11].

3.2 Atomic Formulae (Instance of and Equality)

The predicate ":" (instance of) is translated into the Prolog predicate `instanceof/2`. Which generates the representation of all instances (first argument) of all classes (second argument) of a specification. Thanks to our incomplete structures every instance of a subclass is an instance of a superclass, a technique that makes the desirable property of subsumption to be a theorem in our Prolog axiomatisation.

Clay equality (=) is the other predicate used in the atomic formulae of Clay in this work. Our translation of Clay equality into Prolog consists of two steps: a reduction of the object expressions to normal form and the unification of the obtained representations.

Let us see a description of the implementation of the reduction step and postpone the formalisation of the translation of the equality literals to Section 3.3. Predicate `reduce/2` relates terms that represent abstract syntax trees of Clay expressions with their normal form. The most important clause of `reduce/2` defines the reduction of sending a message (`M`) to an object expression `O`. Functor `.-`, in infix form, represents the send operator of Clay:

```
reduce(O<--M,NF) :- M =.. [Mid|Args],
                    reduce(O,ONF), reduceall(Args,ArgsNF),
                    knownclasses(ONF,Cs),
                    checkpreposts(Cs,ONF,Mid,ArgsNF,NF,defined).
```

```
modifier insert (x : Int) {
 post {
  self : Empty ∧
  result = BSTInt.mkNode(
             x,
             BSTInt.mkEmpty,
             BSTInt.mkEmpty)


  ∨
  self : Node ∧
  (x < self.data : True ∧

   result = BSTInt.mkNode(
              self.data,
              self.left.insert(x),
              self.right)


   ∨ ...) }
}
```

```
post('BSTInt',_s,insert,[_x],_r) :-
 instanceof(_s,'Empty'),
 reduce('BSTInt'<--mkNode(
           _x,
           'BSTInt'<--mkEmpty,
           'BSTInt'<--mkEmpty),
        _NF_BSTInt_mkNode),
 eq('BSTInt',_r,NF_BSTInt_mkNode).
post('BSTInt',_s,insert,[_x],_r) :-
 instanceof(_s, 'Node'),
 reduce(_x < _s<--data,_NF__x_le),
 instanceof(_NF__x_le,'True'),
 reduce('BSTInt'<--mkNode(
           _s<--data,
           _s<--left<--insert(_x),
           _s<--right),
        _NF_BSTInt_mkNode),
 eq('BSTInt', _r, _NF_BSTInt_mkNode).
...
```

Figure 4 Translation of insert.

```
modifier remove (x : Int) {
 post { result.contains(x) : False

      ∧ result.insert(x)=self
 }
}
```

```
post('BSTInt',_s,remove,[_x],_r) :-
 reduce(_r<--contains(_x), _NF__r_contains),
 instanceof(_NF__r_contains, 'False'),
 reduce(_r<--insert(_x), _NF__r_insert),
 eq('BSTInt',_NF__r_insert,_s).
```

Figure 5 Translation of remove.

Predicate `reduceall/2` reduces a list of expressions, the second argument of `knownclasses/2` contains the known classes (`Cs`) of the recipient of the message, and `checkpreposts` checks pre- and post-conditions of every class of `Cs` in which method `Mid` is defined.

We already mentioned in Section 2 the danger of overriding the properties of methods in subclasses: the practical impossibility of reasoning in large programs. The above implementation of predicate `reduce/2` will fail if any postcondition in the inheritance hierarchy is inconsistent with the postconditions specified in superclasses.

3.3 Translation of Pre- and Post-conditions

The translation of formulae takes into account that objects involved in atomic predicates must be reduced. The translation of non-atomic formulae are directly translated into first-order logic formulae resulting in extended programs (sets of implications with an arbitrary first-order formulae in the body). Then, a Lloyd-Topor transformation [17, 18] is applied to obtain a logic program.

Figures 4 and 5 present, in parallel, the correspondence between the Clay specification of methods `insert` and `remove` of `BSTInt` and the automatically synthesised Prolog code.

4 Experimental Results

Let us get back to the problems posed in Section 2.3. Relevant parts of the code obtained for the recursive definition of insert are shown in Figure 4. We have automatically generated up to 4000 trees with up to 80 nodes with a maximum depth of 10. The insertion of elements works properly and the execution time in the worst case is less than 1000 milliseconds.

The second question was whether Clay would be able to generate an executable prototype from the implicit specification of the remove method for binary search trees. The code obtained is shown in Figure 4.

Running tests on this specification shows several problems. First, the logic program obtained from the specification seemed to be only partially correct. Given a (valid) binary insertion tree and one of its elements, the returned tree was, in some cases, a tree meeting the specification but failed in the rest.

Analysis of the tests revealed that the prototype was working properly exactly in those cases where the element to remove was *at the leaves* of the structure — i.e. in a node with two empty subtrees as children. This solves the mystery: the specification for remove uses predefined (structural) equality while the specifier was probably thinking in the intended set semantics for the trees as collections. The order in which elements are stored in the tree affects its actual shape. That is why only elements that make their way down to the "bottom" of the structure via method insert meet the specification of remove.

In other words, the specification was flawed and the execution allowed us to spot the bug. There are several ways to solve the problem. One of them is, of course, to use a *self normalizing* data structure – balanced tree, heap...– for which predefined equality behaves as set equality. A quicker fix – less efficient – is to *flatten* both sides of the equality:

```
modifier remove (x : Int) {
  post {result.contains(x) : False ∧ result.insert(x).flatten() = self.flatten()}
}
```

where flatten is an observer defined recursively in the obvious way:

```
observer flatten () : List <Int> {
  post { self : Empty ∧ result = []
       ∨ self : Node ∧ result = self.left.flatten.append([].cons(self.data)).
                                append(self.right.flatten) }
}
```

We show now the effects of the safe inheritance:

```
?- reduce('Cell'<--mkCellCase(0),R).
R = 'CellCase'{contents : 0}
?- reduce('Cell'<--mkCellCase(0)<--set(1)),R).
R = 'CellCase'{contents : 1}
?- reduce('Cell'<--mkCellCase(0)<--set(1)<--get,R).
R = 1
?- reduce('ReCell'<--mkReCell<--set(0)<--set(1)<--restore<--get,R).
R = 0
?- reduce('Cell'<--mkCell<--set(0)<--set(1)<--restore<--get,R).
no
```

The table below shows performance figures obtained in an Intel Dual Core T7200@2.00GHz, with 4096KB of cache and 2GB of RAM running GNU/Linux 2.6.32-25 SMP and SWI-Prolog v. 5.10.0. The depth limit used for the iterative deepening strategy for predicate instanceof

was 38. The full Clay code for these examples, their translation into Prolog and the Prolog implementation of the Clay theory can be found at `http://babel.ls.fi.upm.es/~angel`.

Test	**Time** (ms.)
Generation of trees (1000 trees)	202
Creation of trees (15 insertions)	929
Removing leaf from tree (1 node)	0
Removing leaf from tree (3 nodes)	391
Removing leaf from tree (7 nodes)	7211
Removing leaf from tree (15 nodes)	18300

5 Related Work and Conclusions

We have presented the compilation scheme of an object oriented formal notation into logic programs. This allows the generation of executable prototypes that help in validating requirements, e.g. by means of automated test generation. We have emphasized the generation of code from implicit method specifications, specially in presence of recursive definitions, something which is seldom supported by other lightweight methods and tools.

Early experiments with our prototype compiler show the feasibility of the approach, but also the limitations of a naive application of Prolog's standard search mechanisms. In fact, obtaining an efficient search scheme is one of the challenges for future research. Our current implementation combines techniques such as the Lloyd-Topor transforms of first-order formulae and iterative deepening search for achieving completeness in some examples.

We expect to increase efficiency with the use of constructive negation and also with techniques that allow for *lazy* instance generation, that is, *coroutining* the logic code that implements quantification via instance generation with the one that implements the implicit postconditions. More mature tools, like ProB [15, 16] already take advantage of these.

One improvement that has already been incorporated in this version is the use of constraints for arithmetic. A previous version used a Peano representation for naturals (predefined class Nat) as a way of obtaining a complete theory for numbers. Now, predefined class Int encapsulates integers that get translated into Prolog integers managed via finite domain constraints. The tests show drastic gains over our previous implementation.

One aspect hard to implement properly is nondeterminism. If the specs are assumed correct, then it suffices to choose one interpretation at random to obtain an executable prototype. This can be achieved, for instance, by limiting nondeterminism in the logic program generated by always choosing the first solution at any choice point.

But if the goal is to use the prototypes for requirement validation, then choosing the good one by chance does not help. In this case, interpretations must be generated randomly, but any of these must be internally consistent, i.e. methods intended to be deterministic must always return the same answer in each interpretation. Ensuring this in Prolog is trickier and can be achieved, for instance, using tabulation techniques.

Certain features of object oriented programming (e.g. mutable state) have been left out of this presentation. Studying the introduction of state in our code generation scheme would help in applying the ideas presented in this paper to other object oriented formal notations like VDM++, Object-Z, Troll or OASIS [7, 23, 14, 19].

References

1 Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
2 Alloy Website. `http://alloy.mit.edu`.
3 Jeffrey Van Baalen and Richard E. Fikes. The role of reversible grammars in translating between representation languages. In *Proceedings of the 4th International Conference on Principles of Knowledge Representation and Reasoning*, pages 562–571, 1994.
4 Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer Verlag, 2004.
5 James L. Caldwell. Extracting general recursive program schemes in Nuprl's type theory. In *LOPSTR '01*, pages 233–244, London, UK, 2001. Springer-Verlag.
6 Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP*, pages 268–279, 2000.
7 John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs For Object-oriented Systems*. Springer-Verlag TELOS, 2005.
8 E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object Oriented Software*. Addison-Wesley, 1995.
9 Vinu George and Rayford Vaughn. Application of lightweight formal methods in requirement engineering. *CrossTalk - The Journal of Defense Software Engineering*, January 2003.
10 Ángel Herranz. *An Object-Oriented Formal Notation: Executable Specifications in Clay*. PhD thesis, Universidad Politécnica de Madrid, January 2011.
11 Ángel Herranz and Julio Mariño. Executable specifications in an object oriented formal notation. In *LOPSTR 2010*, July 2010.
12 D. Jackson and J. Wing. Lightweight formal methods. *IEEE Computer*, April 1996.
13 Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.
14 Ralf Jungclaus, Gunter Saake, Thorsten Hartmann, and Cristina Sernadas. TROLL a language for object-oriented specification of information systems. *ACM Trans. Inf. Syst.*, 14(2):175–211, 1996.
15 Michael Leuschel and Michael Butler. ProB: A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.
16 Michael Leuschel, Dominique Cansell, and Michael Butler. Validating and animating higher-order recursive functions in B. In Jean-Raymond Abrial and Uwe Glässer, editors, *Festschrift for Egon Börger*, 2007.
17 John W. Lloyd and Rodney W. Topor. Making Prolog more expressive. *J. Log. Program.*, 1(3):225–240, 1984.
18 John Wylie Lloyd. *Foundations of Logic Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1993.
19 Oscar Pastor Lopez, Fiona Hayes, and Stephen Bear. *OASIS: An Object-Oriented Specification Language.*, volume 593 of *LNCS*, pages 348–363. Springer, January 1992.
20 Ulf Norell. Dependently typed programming in Agda. In *TLDI '09: Proceedings of the 4th international workshop on Types in language design and implementation*. ACM, 2009.
21 Nicolas Oury and Wouter Swierstra. The power of Pi. *SIGPLAN Not.*, 43(9):39–50, 2008.
22 Prover9 and Mace4 Website. `http://www.cs.unm.edu/%7emccune/mace4`.
23 Graeme Smith. *The Object-Z specification language*. Kluwer Academic Publishers, 2000.