Executable Specifications in an Object Oriented Formal Notation

Ángel Herranz and Julio Mariño

Universidad Politécnica de Madrid
{aherranz|jmarino}@fi.upm.es

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 a number of 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.

1 Introduction

Lightweight formal methods [14, 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 [15]. 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 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 more subtle 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 [15], 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 [23, 5, 4, 22].

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 canonical translation into executable prototypes to a 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 give a formalisation of the synthesis of logic programs from Clay specifications (Section 4) and some examples of the prototypes generated in action (Section 5).

2 Object Oriented Specifications in Clay

Clay is a lightweight evolution of SLAM-SL [10], a large object oriented notation designed to bridge the gap between formal methods and more widespread software engineering processes ([13, 12, 11]). Clay is a *desugared* version of SLAM-SL where the authors focus on formal aspects superficially treated in the cited works.

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 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 [24] 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, overloading not allowed (just method refinement) and no parametric polymorphism.

Figures 1 and 2 contain three examples of Clay specifications that will guide the whole paper. In this section, the examples will help us to present some of the most important and relevant aspects of our notation.

2.1 Modelling Data

Let us start with the specification of natural numbers (Figure 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 n is an instance of Nat (n : Nat) then it is an instance of Zero or, exclusively, of Succ. The following Clay formula expresses it formally:

$$\forall\ \mathsf{n} : \mathsf{Nat}\ ((\mathsf{n} : \mathsf{Zero} \lor \mathsf{n} : \mathsf{Succ}) \land \mathsf{n} : \mathsf{Zero} \Leftrightarrow \neg\ \mathsf{n} : \mathsf{Succ})$$

```
class Nat <: Num {
  state Zero {}
  state Succ {pred : Nat}
  modifier add (n : Nat) {
    post {   self : Zero ⇒ result = n
           ∧ self : Succ ⇒ self←pred←add(n←inc) = result }
  }
  observer even : Bool {
    post { result : True ⇔ exists n : Nat ( self = n←add(n)) }
    sol {    self : Zero ∧ result : True
           ∨ self : Succ ∧ result = self←pred←even←neg }
  }
  observer half : Nat {
    pre { self←even = Bool←mkTrue }
    post { self = result←add(result) }
  }
}
```

Fig. 1. Natural numbers in Clay

The case classes Zero and Succ introduce the constructor methods mkZero and mkSucc. Both are messages that can be sent to the object Nat (classes are objects in Clay): Nat←mkZero[1] or Nat←mkSucc(n), being n an instance of Nat. Let us use 0, 1, 2, etc. to abbreviate Nat←mkZero, Nat←mkSucc(Nat←mkZero), Nat←mkSucc(Nat←mkSucc(Nat←mkZero)), etc.

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 Nat←mkSucc(n)←pred is n.

Inheritance. Classes can be extended with subclasses that inherit all the properties of the superclass. On the left of the Figure 2, class RGBNat extends Nat and therefore its instances obey the aforementioned property:

$$\forall\ \mathsf{n} : \mathsf{RGBNat}\ ((\mathsf{n} : \mathsf{Zero} \vee \mathsf{n} : \mathsf{Succ}) \wedge \mathsf{n} : \mathsf{Zero} \Leftrightarrow \neg\, \mathsf{n} : \mathsf{Succ})$$

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

We consider that 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, on our view, the decision pays back.

[1] In Clay ← is used to indicate method invocation instead of the dot notation used in C++ and other widespread OO programming languages.

```
class RGBNat <: Nat {
  state Red {}
  state Green {}
  state Blue {}

  constructor mkRZ {
    post {     result : Zero
           ∧   result : Rojo }
  }
}

class CMYNat <: Nat {
  state Cyan {}
  state Magenta {}
  state Yellow {}

  constructor mkCZ {
    post {     result : Zero
           ∧   result : Cyan }
}
```

```
class Cell {
  state CellCase { contents : Nat }
  observer get : Nat {
    post { result = self←contents }
  }
  modifier set (v : Nat) {
    post { result ←contents = v }
  }
}

class ReCell <: Cell {
  state ReCellCase { backup : Nat }
  modifier set (v : Nat) {
    post { result ←backup = self←contents }
  }
  modifier restore {
    post {     result ←contents = self←backup
           ∧   result ←backup = self←backup
      }
  }
}
```

Fig. 2. Inheritance in Clay

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). Atomic formulae are equalities and class membership.

Class membership is mainly used to do pattern matching. In the specification of method add we can see how antecedents of implications distinguish the cases zero and non-zero (self : Zero and self : Succ).

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. In the example of add, the minimum type of result and self in the consequent of the first implication is Nat. The semantics establishes that no properties of self other than those *reachable* from Nat are enforced in result .

Let us illustrate it adding a *red* zero to a zero (0←add(RGBNat←mkRZ)). Add establishes that the result is equal, up to Nat, to RGBNat←mkRZ, so the property 0←add(RGBNat←mkRZ) : Zero holds (by the postcondition of mkRZ) and 0←add(RGBNat←mkRZ) : Red is not even a valid expression. Nevertheless, we can write the formula cn = 0←add(RGBNat←mkRZ) on a coloured natural cn : RGBNat since the equality can be indexed with Nat, the minimum common subtype of cn and 0←add(RGBNat←mkRZ).

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.

The specifier can feed the tools with extra executable specifications. The specification of even includes two postconditions (**post** and **sol**). The postcondition can be considered too hard to be used in the prototype and the specifier added a *solution* with the recursive decision procedure. In contrast, observer half is specified by the very natural property $2 \times n \longleftarrow \mathsf{half} = n$.

Cell and ReCell examples on the right of 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.

2.3 Interacting with Clay

In Section 5 we will check the actual results of our prototype to some examples that we present in the following lines:

– Specification of add is an implicit one. Will our prototype be able to compute 42⟵half? What should be the result for 27⟵half, where precondition does not hold?
– The Clay equality is particularly curious. What would be the answers to the following Clay atomic formulae?
  • Nat⟵mkZero = CMYNat⟵mkRZ
  • Nat⟵mkCZ = CMYNat⟵mkRZ
  • Nat⟵mkZero⟵inc = CMYNat⟵mkRZ⟵inc
– Finally, to check that we are enabling the specifier to write concise specifications, what will be the answers to expressions like
  ReCell⟵mkReCell⟵set(0)⟵set(1)⟵get and
  ReCell⟵mkReCell⟵set(0)⟵set(1)⟵restore⟵get?

3 Translating Clay Specifications into Logic Programs

Given a Clay specification we will synthesise facts that represent its abstract syntax tree: classes, inheritance, case classes, fields, and method's pre and postconditions. In Section 4 we give a formalisation of the translation, for the moment Figure 3 contains the informal meaning of the target predicates.

The heart of our prototype 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. Their meaning:

– Predicate instanceof(NF,A) is a generator of instances NF of a class A. NF is a normal form of an instance of A: a flexible representation as an incomplete data structure that we will study in Section 3.1.

`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 cases classes of class C
`fields(`C`,`Fs`)`	Fs is the association list with the field names and field types of the case class C
`msgtype(`C`,`M`)`	M is a message identifier of a method defined or overridden in class C
`pre(`C`,`S`,`M`,`As`)`	Precondition of sending message M with arguments As to instance S in 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 in class C

Fig. 3. Representing Clay in Prolog

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

3.1 Representing Clay Instances in Prolog

How will we represent the natural number 1 (an instance of Nat)? We need to capture all the information of known superclasses (Num) and to capture all the information about the specific case class (Nat). 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 of field names to the representation of the instances of their classes. Let us show the representation of number 1:

```
[('Num','Num',[]), ('Nat','Succ',[(pred,[('Num','Num',[]),
                                         ('Nat','Zero',[])])])]
```

Under subtyping, during a deduction process where 1 is expected the successor of a *red 0* could appear. If we follow our rules, the representation of the coloured number would be:

```
[('Num','Num',[]), ('Nat','Succ',[(pred,[('Num','Num',[]),
                                         ('Nat','Zero',[]),
                                         ('RGBNat','Red',[])])])]
```

The representation of 0 and red 0 are partially the same:

```
[('Num','Num',[]),('Nat','Zero',[])]
[('Num','Num',[]),('Nat','Zero',[]),('RGBNat','Red',[])]
```

We propose to *make room* for not yet known information of subclasses. Our proposal is to use an incomplete data structure where the incomplete part represents the *room* for the information of potential subclasses of `Nat`:

```
[('Num','Num',[]),('Nat','Zero',[]),_Room]
[('Num','Num',[]),('Nat','Zero',[]),('RGBNat','Red',[]),_Room]
```

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

– Information of the 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 RGBNat) 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 a red 0 *fits* in a 0.

3.2 Instance of

Predicate ":" (instance of) is translated into the Prolog predicate `instanceof/2`. `Instanceof/2` generates the representation of *all* instances (first argument) of all classes (second argument) of a specification. Let us see some outputs of this predicate:

```
?- instanceof(O,C).
C = 'Num', O = [('Num','Num',[])|_] ? ;
C = 'Bool', O = [('Bool','True',[])|_] ? ;
...
C = 'RGBNat',
O = [('Num','Num',[]),('Nat','Zero',[]),('RGBNat','Red',[])|_] ? ;
C = 'RGBNat',
O = [('Num','Num',[]),('Nat','Zero,[]),('RGBNat','Green',[])|_]␣?
```

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.

3.3 Equality

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 to normal form of the objects expressions 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 4. `Reduce/2` relates terms that represent abstract syntax trees of Clay expressions with its normal form (assume `:- op(200,yfx,'<--').` declared to emulate the Clay syntax):

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

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

Safe Inheritance. 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.

4 The Translation, Formalised

In this section we show an abstract representation of specifications in Clay. A specification (*Spec*) is a partial function from class identifiers (*CI*) to class specifications (*CS*). A class specification is a triple with the identifier of the superclass, a state environment (*SE*) and a method environment (*ME*). A state environment is a partial function from class identifiers (the case classes) to field environments (*FE*). A field environment is a partial function from method identifiers (*MI*) to class identifiers. A method environment is a partial function from method identifiers (*MI*) to method specifications (*MS*). A method specification is a triple with a method declaration (*MD*), and two formulae (*Form*) that represent the precondition and the postcondition. A method declaration is a partial function from object variables (*OV*) to class identifiers.

Figure 4 contains the mathematical definition of the abstract syntax. We use the following conventions in our metalanguage: A and B for class identifiers (*CI*), e, a and b for object expressions (*Expr*), x for object variables (*OV*), m for method identifiers (*MI*), and F and G for formulae (*Form*).

$$\begin{array}{lrl}
\textbf{Formulae}\,(F, G) & \mathit{Form} ::= & e_1 = e_2 \mid e : A \\
 & & \mid \neg F \mid F \wedge G \mid F \vee G \mid F \Rightarrow G \mid F \Leftrightarrow G \\
 & & \mid \forall x : A(F) \mid \exists x : A(F) \\
\textbf{Expressions}\,(a, b, e) & \mathit{Expr} ::= & A \mid x \mid e \leftarrow m(e_1, e_2, \ldots e_n) \\
\textbf{Specifications} & \mathit{Spec} = & \mathit{CI} \nrightarrow \mathit{CS} \\
\textbf{Class Specifications} & \mathit{CS} = & \mathit{CI} \times \mathit{SE} \times \mathit{ME} \\
\textbf{State Environments} & \mathit{SE} = & \mathit{CI} \nrightarrow \mathit{FE} \\
\textbf{Field Environments} & \mathit{FE} = & \mathit{MI} \nrightarrow \mathit{CI} \\
\textbf{Method Environments} & \mathit{ME} = & \mathit{MI} \nrightarrow \mathit{MS} \\
\textbf{Method Specifications} & \mathit{MS} = & \mathit{MD} \times \mathit{Form} \times \mathit{Form} \\
\textbf{Method Declarations} & \mathit{MD} = & \mathit{OV} \nrightarrow \mathit{CI}
\end{array}$$

Fig. 4. Clay's abstract syntax

Figure 6 shows the translation of Clay specifications, into Prolog programs (*Prog*), although extended programs (*EG*, *EC*, *EP*) allowing general first-order constructs in goals are used as an intermediate format and then translated into

Terms (s, t)	Term	$::=$	$x \mid f(t_1, \dots, t_n)$
Atoms (A, B)	Atom	$::=$	$p(t_1, \dots, t_n)$
Literals (L)	Lit	$::=$	$A \mid \neg A$
Goals (G)	Goal	$=$	Lit^*
Horn Clauses (C)	Clause	$=$	$\mathit{Atom} \times \mathit{Goal}$
Programs (P, Q)	Prog	$=$	$\mathbb{P}\,\mathit{Clause}$
Extended Goals(F)	EG	$::=$	$\mathit{Lit} \mid F \wedge F' \mid F \vee F' \mid F \Rightarrow F' \mid F \Leftrightarrow F'$
			$\mid \forall\, x : A(F) \mid \exists\, x : A(F)$
Ext. Clauses(K)	EC	$=$	$\mathit{Atom} \times \mathit{EG}$
Ext. Programs	EP	$=$	$\mathbb{P}\,\mathit{EC}$

Fig. 5. Prolog's abstract syntax

standard logic programs via a Lloyd-Topor like transform ([20, 19]). A concise abstract syntax of logic programs is given in Figure 5. The part of translation dealing with formulae and expressions is shown in Figure 7. Due to space limitations, the definitions of several auxiliary functions are given informally.

5 Experimental Results

Let us show some details of the code generated for the examples in our test set.

Recursive definitions. Our implementation of the Lloyd-Topor transformation applied to the postcondition of `add` produces four clauses:

```
post('Nat', _self, add, [_n], _result) :-
   instanceof(_result, 'Nat'),
   \+ instanceof(_self, 'Zero'),
   \+ instanceof(_self, 'Succ').
post('Nat', _self, add, [_n], _result) :-
   instanceof(_result, 'Nat'),
   \+ instanceof(_self, 'Zero'),
   reduce('Nat'<--mkSucc(_self<--pred<--add(_n)), _NF_Nat_mkSucc),
   eq('Nat', _result, _NF_Nat_mkSucc).
post('Nat', _self, add, [_n], _result) :-
   instanceof(_result, 'Nat'),
   eq('Nat', _result, _n),
   \+ instanceof(_self, 'Succ').
post('Nat', _self, add, [_n], _result) :-
   instanceof(_result, 'Nat'),
   eq('Nat', _result, _n),
   reduce('Nat'<--mkSucc(_self<--pred<--add(_n)), _NF_Nat_mkSucc),
   eq('Nat', _result, _NF_Nat_mkSucc).
```

Since not enough intelligence has been imprinted, the first and the last clause are, respectively, the result of falsifying the antecedents of implications and checking the consequents. None of those clauses yield any new result in our case: `self` cannot be and instance of `Zero` and an instance of `Succ` and the last

$\mathsf{tr\_spec} : \mathit{Spec} \to \mathit{Prog}$
$\mathsf{tr\_spec}[\![S]\!] = \mathsf{Clay\_Theory} \cup \bigcup_{A \in \mathrm{dom}\, S} \mathsf{lloyd\_topor} \circ \mathsf{tr\_class}[\![A, S(A)]\!]$

$\mathsf{lloyd\_topor} : \mathit{EP} \to \mathit{Prog}$
$\mathsf{lloyd\_topor}[\![ep]\!] =$ The Lloyd-Topor Transformation ([20, 19])

$\mathsf{tr\_class} : \mathit{CI} \times \mathit{CS} \to \mathit{EP}$
$\mathsf{tr\_class}[\![A, (\mathit{super}, \mathit{states}, \mathit{methods})]\!] = \{(\texttt{class(}\mathsf{tr\_exp}[\![A]\!]\texttt{)}, \top)$
$\quad ,(\texttt{instanceof(}\mathsf{tr\_exp}[\![A]\!]\texttt{,}\mathsf{tr\_meta}[\![A]\!]\texttt{)}, \top)$
$\quad ,(\texttt{inherits(}\mathsf{tr\_exp}[\![A]\!]\texttt{,}\mathsf{tr\_exp}[\![\mathit{super}]\!]\texttt{)}, \top)\}$
$\quad \bigcup \mathsf{tr\_se}[\![A, \mathit{states}]\!] \bigcup \mathsf{tr\_me}[\![A, \mathit{methods}]\!]$

$\mathsf{tr\_se} : \mathit{CI} \times \mathit{SE} \to \mathit{EP}$
$\mathsf{tr\_se}[\![A, \mathit{states}]\!] = \{(\texttt{cases(}\mathsf{tr\_exp}[\![A]\!]\texttt{,}\mathsf{tr\_set}[\![\mathrm{dom}\, \mathit{states}]\!]\texttt{)}, \top)\}$
$\quad \bigcup_{A \in \mathrm{dom}\, \mathit{states}} (\mathsf{tr\_stt}[\![A, B, \mathit{states}(B)]\!] \cup \mathsf{tr\_fe}[\![A, B, \mathit{states}(B)]\!])$

$\mathsf{tr\_stt} : \mathit{CI} \times \mathit{CI} \times \mathit{FE} \to \mathit{EP}$
$\mathsf{tr\_stt}[\![A, B, \mathit{fields}]\!] = \{(\texttt{msgtype(}\mathsf{tr\_exp}[\![A]\!]\texttt{,mk+}\mathsf{tr\_exp}[\![B]\!]\texttt{)}, \top)$
$\quad , (\texttt{pre(}\mathsf{tr\_meta}[\![A]\!]\texttt{,\_self,mk+}\mathsf{tr\_exp}[\![B]\!]\texttt{,}\mathsf{tr\_fv}[\![\mathit{fields}]\!]\texttt{)},$
$\qquad \texttt{instanceof(\_self,}\mathsf{tr\_meta}[\![A]\!]\texttt{)}$
$\qquad \wedge\ \mathsf{tr\_fety}[\![\mathit{fields}]\!])$
$\quad , (\texttt{post(}\mathsf{tr\_meta}[\![A]\!]\texttt{,\_self,mk+}\mathsf{tr\_exp}[\![B]\!]\texttt{,}\mathsf{tr\_fv}[\![\mathit{fields}]\!]\texttt{,\_result)},$
$\qquad \texttt{instanceof(\_result,}\mathsf{tr\_meta}[\![B]\!]\texttt{)}$
$\qquad \wedge\ \texttt{project(\_result,}\mathsf{tr\_exp}[\![B]\!]\texttt{,}\mathsf{tr\_fenv}[\![\mathit{fields}]\!]\texttt{)})$

$\mathsf{tr\_fe} : \mathit{CI} \times \mathit{CI} \times \mathit{FE} \to \mathit{EP}$
$\mathsf{tr\_fe}[\![A, B, \mathit{fields}]\!] = \bigcup_{m \in \mathrm{dom}\, \mathit{fields}} \mathsf{tr\_field}[\![A, B, \mathit{fields}(m), i, m]\!]$

$\mathsf{tr\_field} : \mathit{CI} \times \mathit{CI} \times \mathit{CI} \times \mathit{MI} \times \mathbb{N} \to \mathit{EP}$
$\mathsf{tr\_field}[\![A, B, C, i, \mathit{fn}]\!] = \{(\texttt{msgtype(}\mathsf{tr\_exp}[\![A]\!]\texttt{,}\mathsf{tr\_mi}[\![\mathit{fn}]\!]\texttt{)}, \top)$
$\quad , (\texttt{pre(}\mathsf{tr\_meta}[\![A]\!]\texttt{,\_self,}\mathsf{tr\_mi}[\![\mathit{fn}]\!]\texttt{,[])},$
$\qquad \texttt{instanceof(\_self,}\mathsf{tr\_meta}[\![B]\!]\texttt{)})$
$\quad , (\texttt{post(}\mathsf{tr\_meta}[\![A]\!]\texttt{,\_self,}\mathsf{tr\_mi}[\![\mathit{fn}]\!]\texttt{,[],\_result)},$
$\qquad \texttt{instanceof(\_result,}\mathsf{tr\_meta}[\![C]\!]\texttt{)}$
$\qquad \wedge\ \texttt{project(\_self,}\mathsf{tr\_exp}[\![B]\!]\texttt{,}\mathsf{tr\_anon}[\![i, \mathit{fn}, C]\!]\texttt{)})$

$\mathsf{tr\_me} : \mathit{CI} \times \mathit{ME} \to \mathit{EP}$
$\mathsf{tr\_me}[\![A, B, \mathit{methods}]\!] = \bigcup_{m \in \mathrm{dom}\, \mathit{methods}} \mathsf{tr\_ms}[\![A, \mathit{methods}(m)]\!]$

$\mathsf{tr\_ms} : \mathit{CI} \times \mathit{MI} \times \mathit{MS} \to \mathit{EP}$
$\mathsf{tr\_ms}[\![A, m, (\mathit{md}, \mathit{pre}, \mathit{post})]\!] = \mathsf{tr\_md}[\![A, m, \mathit{md}]\!] \cup$
$\quad \{(\texttt{pre(}\mathsf{tr\_exp}[\![A]\!]\texttt{,\_self,}\mathsf{tr\_mi}[\![m]\!]\texttt{,}\mathsf{tr\_args}[\![\mathit{md}]\!]\texttt{)},$
$\qquad \mathsf{tr\_form}[\![\mathit{pre}]\!])$
$\quad , (\texttt{post(}\mathsf{tr\_exp}[\![A]\!]\texttt{,\_self,}\mathsf{tr\_mi}[\![m]\!]\texttt{,}\mathsf{tr\_args}[\![\mathit{md}]\!]\texttt{,\_result)},$
$\qquad \mathsf{tr\_form}[\![\mathit{post}]\!])$

$\mathsf{tr\_md} : \mathit{CI} \times \mathit{MD} \to \mathit{EP}$
$\mathsf{tr\_md}[\![A, m]\!] = \{(\texttt{msgtype(}\mathsf{tr\_exp}[\![A]\!]\texttt{,}\mathsf{tr\_mi}[\![m]\!]\texttt{)}, \top)\}$

tr_meta	generates the class a given class A is instance of: `meta`A
tr_set	generates a Prolog list with valid Prolog terms that represent the set of class identifiers
tr_mi	generates a valid Prolog term to represent a message identifier

Fig. 6. Translating Clay specifications

$\mathsf{tr\_form} : \mathit{Form} \to \mathit{EG}$
$\mathsf{tr\_form}\llbracket e_1 = e_2 \rrbracket$ $=$ `reduce(`$\mathsf{tr\_exp}\llbracket e_1 \rrbracket$`, _NF1)` $\wedge$ `reduce(`$\mathsf{tr\_exp}\llbracket e_2 \rrbracket$`, _NF2)`
$\wedge$ `eq(`$\mathsf{tr\_exp}\llbracket A \rrbracket$`, _NF1, _NF2)`
where A is the minimum common type of e_1 and e_2
and `_NF1` and `_NF2` are new variables
$\mathsf{tr\_form}\llbracket e : A \rrbracket$ $=$ `reduce(`$\mathsf{tr\_exp}\llbracket e \rrbracket$`, _NF)` $\wedge$ `instanceof(_NF, `$\mathsf{tr\_exp}\llbracket A \rrbracket$`)`
where `_NF` is a new variable
$\mathsf{tr\_form}\llbracket \neg F \rrbracket$ $=$ `negate(`$\mathsf{tr\_form}\llbracket F \rrbracket$`)`
$\mathsf{tr\_form}\llbracket F * G \rrbracket$ $= \mathsf{tr\_form}\llbracket F \rrbracket * \mathsf{tr\_form}\llbracket G \rrbracket$
where $* \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$
$\mathsf{tr\_form}\llbracket \forall\, x : A(F) \rrbracket = \forall\, \mathsf{tr\_exp}\llbracket x \rrbracket (\mathsf{tr\_form}\llbracket x : A \rrbracket \Rightarrow \mathsf{tr\_form}\llbracket F \rrbracket)$
$\mathsf{tr\_form}\llbracket \exists\, x : A(F) \rrbracket = \exists\, \mathsf{tr\_exp}\llbracket x \rrbracket (\mathsf{tr\_form}\llbracket x : A \rrbracket \wedge \mathsf{tr\_form}\llbracket F \rrbracket)$

$\mathsf{tr\_exp} : \mathit{Expr} \to \mathit{Term}$
$\mathsf{tr\_exp}\llbracket A \rrbracket = \mathsf{mk\_const}\llbracket A \rrbracket$
$\mathsf{tr\_exp}\llbracket x \rrbracket = \mathsf{mk\_var}\llbracket x \rrbracket$
$\mathsf{tr\_exp}\llbracket e \leftarrow m(e_1, \ldots, e_n) \rrbracket =$ `send(`$\mathsf{tr\_exp}\llbracket e \rrbracket$`,`$\mathsf{tr\_mi}\llbracket m \rrbracket$`(`$\mathsf{tr\_exp}\llbracket e_1 \rrbracket$`,`$\ldots$`,`$\mathsf{tr\_exp}\llbracket e_n \rrbracket$`)`

tr_fv	generates a list with a variable per field
tr_fety	generates `instanceof` calls given a list of vars and a list of classes
tr_fenv	generates a partial function from field names to vars
tr_args	generates a list of vars per arg in a message

Fig. 7. Translating Clay formulae and expressions

clause is establishing that result $=$ n and result $=$ self $+$ n forcing self to be 0 and repeating the results of third clause. The result of some messages[2]:

```
?- reduce('Nat'<--mkZero,_Zero), reduce('Nat'<--mkSucc(_Zero),_One),
   reduce(_Zero<--add(_One),_One), reduce(_One<--add(_Zero),_One),
   reduce('Nat'<--mkSucc(_One),_Two), reduce(_Two<--add(Two),_Four).
Zero = Zero{},
One = Succ{pred : Zero{}},
Two = Succ{pred : Succ{pred : Zero{}}}
Four = Succ{pred : Succ{pred : Succ{pred : Succ{pred : Zero{}}}}}?
```

Our next example is the translation of half. For the moment we can see that the synthesised code is structurally the same that the specified in Clay:

```
post('Nat', _self, half, [], _result) :-
   instanceof(_result, 'Nat'),
   reduce(_result<--add(_result), _NF__result_add),
   eq('Nat', _self, _NF__result_add).
```

Will our prototype find the resulting instance of `42`$\leftarrow$`half`? And, what about `27`$\leftarrow$`half`?

```
?- num2nat(42,_E42), num2nat(21,_E21),
   reduce(_E42<--half,NF21), reduce(_E21,NF21).
NF21 = [('Num','Num',[]),('Nat','Succ',[(pred,[...])])] ?
```

[2] Prolog terms that represent normal forms are presented in a more readable format.

```
yes
?- num2nat(27,_E27), reduce(_E27<--half,_).
no
```

`Yes`[3], is the answer to 42⟵half with variable `NF21` representing the normal form of our natural number 21. Since precondition of 27⟵half (27⟵even : True) does not hold our prototype's answer is `no`.

Equality. Let us show the answers to queries about the equality of several coloured and non-coloured naturals:

```
?- reduce('RGBNat'<--mkZeroRed,ZR),
   reduce('CMYNat'<--mkZeroCyan,ZC), eq('Nat',ZR,ZC).
ZC = [('Num','Num',[]),('Nat','Zero',[]),('CMYNat','Cyan',[])|_],
ZR = [('Num','Num',[]),('Nat','Zero',[]),('RGBNat','Red',[])|_] ?
?- reduce('Nat'<--mkZero,Z)
   reduce('CMYNat'<--mkZeroCyan,ZC), eq('Nat',Z,ZC).
Z = [('Num','Num',[]),('Nat','Zero',[])|_],
ZC = [('Num','Num',[]),('Nat','Zero',[]),('CMYNat','Cyan',[])|_] ?
```

Overriding. We show now the effects of the safe inheritance:

```
?- reduce('Cell'<--mkCellCase(0),R).
R = 'CellCase'{contents : 'Zero'{}}
?- reduce('Cell'<--mkCellCase(0)<--set('Nat'<--mkSucc(0)),R).
R = 'CellCase'{contents : 'Succ'{pred : 'Zero'{}}}
?- reduce('Cell'<--mkCellCase(0)<--set(1)<--get,R).
R = 'Succ'{pred : 'Zero'{}}
?- reduce('ReCell'<--mkReCellCase(0)<--set(1)<--get,R).
R = 'Succ'{pred : 'Zero'{}}
?- reduce('ReCell'<--mkReCellCase(0)<--set(1)<--restore<--get,R).
R = 'Zero'{}
?- reduce('Cell'<--mkCellCase(0)<--set(1)<--restore<--get,R).
no
```

Performance. We finish the presentation of our results with a pair of performance figures. Our experiments have been produced in a Ubuntu box running GNU/Linux 2.6.31-21 SMP on a machine with an Intel Dual Core CPU T7200@2.00GHz, 4096KB of cache and 2 GB of RAM. Our Prolog engine is Ciao 1.13.0-11293. In the following table we show the performance of our prototypes. Column *depth* indicates the limit in the depth for the iterative deepening strategy for predicate `instanceof`.

Test	**Depth**	**Time** (secs)
Generation of 1000 natural numbers	6000	1.5
`42⟵even`	1000	20.0

The Clay specifications of `Bool`, `Num`, `Nat`, `RGBNat`, `CYMNat`, `Cell`, `ReCell`, their translation into Prolog and the Prolog implementation of the Clay theory, can be found at http://babel.ls.fi.upm.es/~angel/papers/2010lopstr-code.tgz.

[3] Predicate `num2nat/2` translates Prolog positive integers into Clay abstract syntax trees that represent such numbers.

6 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 can help in validating requirements, e.g. by means of testing.

In this paper we have focused on 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 challenging aspects of this work. Our current implementation combines several techniques: Lloyd-Topor transforms of first-order formulae, constructive negation, iterative deepening search, to name a few.

Comparison between the Prolog code obtained from our compiler and that crafted by hand still shows room for improvement. However, there exist more mature tools (see, for instance, ProB [17, 18]) which also generate logic programs from formal specifications, that show that certain extensions of logic programming (constraints, coroutining, etc.) can help in dramatically improving the efficiency of the resulting code in an automated way.

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, 25, 16, 21].

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, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.
4. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.
5. James L. Caldwell. Extracting general recursive program schemes in Nuprl's type theory. In *LOPSTR '01: Selected papers from the 11th International Workshop on Logic Based Program Synthesis and Transformation*, 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, Santa Clara, CA, USA, 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 and Juan José Moreno-Navarro. On the design of an object-oriented formal notation. In *Fourth Workshop on Rigorous Object Oriented Methods, ROOM 4*. King's College, London, March 2002.
11. Ángel Herranz and Juan José Moreno-Navarro. Formal extreme (and extremely formal) programming. In Michele Marchesi and Giancarlo Succi, editors, *4th International Conference on Extreme Programming and Agile Processes in Software Engineering, XP 2003*, number 2675 in LNCS, pages 88–96, Genova, Italy, May 2003.
12. Ángel Herranz and Juan José Moreno-Navarro. Rapid prototyping and incremental evolution using SLAM. In *14th IEEE International Workshop on Rapid System Prototyping, RSP 2003)*, San Diego, California, USA, June 2003.
13. Ángel Herranz and Juan José Moreno-Navarro. *Design Pattern Formalization Techniques*, chapter Modeling and Reasoning about Design Patterns in SLAM-SL. IGI Publishing, March 2007. Other ISBN: 978-1-59904-221-3.
14. D. Jackson and J. Wing. Lightweight formal methods. *IEEE Computer*, pages 21–22, April 1996.
15. Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006.
16. 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.
17. 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.
18. 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.
19. John W. Lloyd and Rodney W. Topor. Making Prolog more expressive. *J. Log. Program.*, 1(3):225–240, 1984.
20. John Wylie Lloyd. *Foundations of Logic Programming.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1993.
21. Oscar Pastor Lopez, Fiona Hayes, and Stephen Bear. *OASIS: An Object-Oriented Specification Language.*, volume 593 of *Lecture Notes in Computer Science*, pages 348–363. Springer, Berlin, Heidelberg, January 1992.
22. Ulf Norell. Dependently typed programming in Agda. In *TLDI '09: Proceedings of the 4th international workshop on Types in language design and implementation*, pages 1–2, New York, NY, USA, 2009. ACM.
23. Nicolas Oury and Wouter Swierstra. The power of Pi. *SIGPLAN Not.*, 43(9):39–50, 2008.
24. Prover9 and Mace4 Website. http://www.cs.unm.edu/%7emccune/mace4.
25. Graeme Smith. *The Object-Z specification language.* Kluwer Academic Publishers, Norwell, MA, USA, 2000.