Verifying BON models with Alloy

Ramiro Adrian Demasi
Dpto. de Computación, Universidad Nacional de Río Cuarto
Río Cuarto, CP. 5800, Argentina
and
Pablo Daniel Ponzio
Dpto. de Computación, Universidad Nacional de Río Cuarto
Río Cuarto, CP. 5800, Argentina
and
Pablo Francisco Castro
Department of Computing and Software, McMaster University
Hamilton, CP. L8S 4K1, Canada
and
Gabriel Baum
LIFIA - Universidad Nacional de La Plata
La Plata, CP. 1900, Argentina

ABSTRACT

In this paper we describe a methodology to translate BON (architectural) designs to Alloy specifications. The main virtue of this process is that it can be implemented by means of software tools. The utilization of this methodology during the software development allows designers to validate different kinds of properties over their BON models. Allowing, in this way, the finding of critical bugs in earlier steps of system construction. Finally, we present a software which implements this translation from BON to Alloy.

Keywords: BON, Alloy, Formal Methods, Software Engineering, Object Oriented Languages, Architectural Models.

1. INTRODUCTION

BON [**?**, **?**] (Business Object Notation) is an object-oriented design language, its principal characteristics are: it has integrated a process of development and it allows designers to describe restrictions by means of assertions, which is particularly relevant to express the behavior of classes.
The present work is based on the utilization of assertions as a way to validate design properties. With this goal in mind, we developed a methodology, based on [**?**], which allows to translate BON models to Alloy specifications [**?**, **?**]. In this way, the resulting formal model is used for checking out different kinds of properties, for example: the model consistency. We have to remark that this task can be done using the Alloy Analyzer tool. We have implemented the translation from BON to ALLOY in a software program called Darwin Tool, developed by the authors of this paper.
This article is structured as follows. In sections 2 and 3 we present a brief introduction to BON and ALLOY, respectively. In section 4 we describe in detail the translation between both languages. In section 5 we show an example of application of this work. Finally, we describe the conclusions and further work.

2. THE BON NOTATION

BON is a language of modeling for specifying and describing software systems. One of its most important characteristics is that it is not only a graphical language to describe systems, additionally it provides a process for developing software. Another important feature of BON is that the graphical language has a equivalent textual notation.
The Business Object Notation is simple compared to other languages [**?**]: it has only two kind of diagrams, and it has not different views of a design, which (sometimes) avoids the inconsistencies introduced when different descriptions of a component are allowed.
BON was designed to support three principal techniques: *Seamlessness*, *Reversibility* and *Software Contracting*.

- Seamlessness: It is the principle of using a consistent set of concepts and notations during the software development, starting in the

analysis step and finishing in the implementation. In this way, the connection between each step is straightforward.

- Reversibility: This means that the changes done in a development step can be directly introduced in early steps. For instance, a change on a Eiffel class is directly reflected in changes on the correspondent design class.

- Design by Contracts [**?**]: It is a methodology to develop object-oriented software, which is based on the concept of contract. A contract is a specification of obligations and requirements of a software component, expressed by means of pre/post-conditions in the Hoare/Floyd style [**?**]. Which allows designers to produce high quality systems.

In the same way that other object-oriented languages, one of the main mechanisms for building models in BON is the notion of *class*. A BON class introduces a new type and it is considered a module. The BON classes are composed by: a name, a set of *features* and a contract made of one precondition, one postcondition and one class invariant. The class invariant is an assertion which all the class instances have to satisfy. In fact, the contract determines the class behavior, and it is expressed using the BON assertion logic, which is a first order logic with types.
The *features* in a class can be one of the following kinds:

- Queries: These are the functions and attributes of classical object-oriented programming. It is important to remark that *queries* does not introduce changes in the system state.

- Commands: It is a name for the concept of procedure, as usually, these change the system state.

Each *feature* has associated a visibility restriction, and it may have different implementations, it could be abstract, concreted or redefined.
Note that BON only support two types of relations inside of a class diagram:

- Client-Supplier relationship: This relationship points out that a class (the client) uses some service provided by the supplier. there are three types of relations Client-Supplier in BON: association, aggregation and shared association.

- Inheritance relationship: By means of this relationship a class inherit behavior from another class. The inheritance relationship is, in fact, a refinement relation between the contracts of both classes involved. Furthermore, in BON, a subclass is a subtype of its superclass.

3. THE ALLOY SPECIFICATION LANGUAGE

Alloy is a formal language, based on first order relational logic. It allows us to specify, in a mathematical way, software systems; specifying the state space, the restrictions and properties of these. An Alloy model is made of *signatures*, *facts*, *predicates*, *restrictions* and *assertions*.
A *signature* introduces a set of atoms, and a type. Additionally, in its declaration is possible to define relations between different *signatures*. For example:

```
sig Person { ID: Int }
```

which define the type *Person*, and a binary relation *ID*. which associates each person with his identity number. Relations could have an arbitrary arity, and we can put restrictions on they cardinality.

```
sig Person { ID: Int,
             works_in: set Company }

sig Company { employees: set Person,
              id: employees -> one Int }
```

where the relation *works_in*: $Company \times Company$ related one person with the company where his works. On the other hand, *id*: $Company \times Employees \times Int$ is a ternary relation which assign an integer to each employee.
In Alloy, the sets associated with different signatures are disjoint, excepting when a signature is declared as an extension of the other signature. This can be done as follows: we suppose that, in the example given at the beginning, we want to distinguish between persons who works in different companies.

```
sig Person { ID: Int }

sig Employee extends Person {
              works_in: set Company }

sig Company { employees: set Employee,
              id: employees -> one Int }
```

In this scenario the signature *Employee* is a subset of the set associated to *Person* and a new type.
In Alloy, a *fact* is a formulae used for restricting the possible values taken by variables. For example, if we want to express that two different persons cannot have the same ID:

```
fact singleID {
  all p1, p2: Person |
      p1 != p2 => p1.ID != p2.ID
}
```

Furthermore, we can add a restriction expressing that a person is related with a company by means of *works_in* if only if he belongs to the set of company employees.

```
fact {
  all emp: Employee | all e: Company |
    e in emp.works_in <=>
      emp in e.employees
}
```

In these formulas the (.) operator denotes the relational composition, and the keyword *in* represents the set inclusion.
Additionally we can define *functions* and *predicates*: *functions* are parametrized formulas which return a value, which could be used in different contexts, replacing their formal parameters by the actual ones. Meanwhile, *predicates* are classical logical formulas with parameters. For example, the following is a predicate expressed in Alloy:

```
pred Fire (e, e': Company,
               emp: e.employees,
               emp': Employee) {
  e'.employees = e.employees - emp
  e'.id = e.id - emp -> (emp.(e.id))
  emp'.works_in = emp.works_in - e
  emp'.ID = emp.ID  }
```

which formalizes the action of firing a employee. Here the primed variables *e'* and *emp'* point out the state of Company after of executing the operation.
Finally, in an Alloy model we can define *assertions*. That is, claims about our model that we hope to be valid, or implied by the model restrictions. For example, if we fire a employee then should be true that he does not belong to the company.

```
assert fired {
  all e1,e2: Company |
  all emp1:e.employees |
  all emp2: Employee |
    Fire(e1, e2, emp1, emp2) =>
      (not (e2 in emp2.works_in) &&
       not (emp2 in e2.employees))
}
```

Alloy has two kinds of commands which may be executed on models:

- Check *assert* for n: It looks for an counterexample for the *assert*, instancing each signature with at most n atoms.
- Run f for n: it looks for a model for the function (or predicate) f, where n is a bound for the model.

We can remark that there exist a software tool for checking Alloy specifications: The Alloy Analyzer, this tool allows us to find models and to run the commands described before, in this way it is possible to verify several properties on our models.
Note that if Alloy Analyzer does not found a counterexample of an assertion this not implies that this property is no valid, it only means that in the bounded models there are not counterexamples. But in the practice it is useful for several practical applications.

4. THE TRANSLATION FROM BON TO ALLOY

In this section we show a translation from BON designs to Alloy specifications. We use the following example to describe the translation:

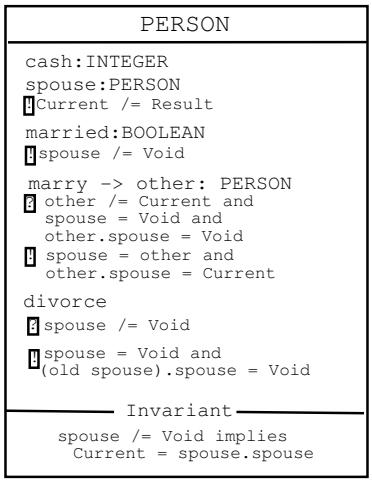


Figure 1: The Person class

4.1 Classes and states

Given a BON design, the first step in the translation is to express the types in the design to signatures in an Alloy specification. To do this we introduce a signature for each class in the design. In the example given before, for the class *PERSON* we introduce:

```
sig Person { }
```

This signature does not have any structure, it will be described in a more general signature called *State*. Informally, *State* describes the system state. That is, the signature *State* simulates the concept of state in Alloy. For instance, if we have the following classes: A, B and C. Then the possible states are described as follows:

```
sig State {
  as: set A,
  bs: set B,
  cs: set C
}
```

Informally, *as*, *bs* y *cs* are the potential instances of A, B y C, respectively.

4.2 Terms and formulas

The correspondence between the formulas in both language is simple. Operators are translate as is described by the following table:

BON	Alloy
and	&&
or	||
implies	=>
iff	<=>
not	!
$x : T$	$x : T$
$x.r$	$x.r$

The terms like *t.query*, which denote a navigation, are traduced with respect a particular state. In this way, the term given before is traduced as: *t.(s.query)* being *s* a variable of *State*.

4.3 Queries

For each not boolean *query* of any class a relation in *State* must be added, which associates objects of this class with the correspondent value in the state. In the example given before the translation is:

```
sig Person { }

sig State {
  Persons: set Person,
  Person_cash: Persons -> Int,
  Person_spouse: Persons -> lone Persons
}
```

Note that each element of type Person must be related with a Int which represents the money of this person, but it not necessarily has a spouse. It is since *spouse* is a reference and *cash* is not. Additionally, for each not boolean *query* it is added a predicate expressing its precondition, another predicate expressing its postcondition and a *fact* to restrict the possible values of the query. Returning to the previous example, for the *query spouse* is added:

```
pred Pre_Person_spouse(s: State,
                       self: s.Persons) {
}

pred Post_Person_spouse(s: State,
                        self: s.Persons,
                        result: s.Persons) {
  self /= result
}

fact {
       all s: State |
       all self: s.Persons |
         Pre_Person_spouse(s, self) &&
         Post_Person_spouse(s, self,
             self.(s.Person_spouse))
}
```

We have to distinguish when a *query* is of Boolean type, since Alloy does not provides this type. We can add this type, with the classical operations on it. The problem is that increases the complexity of the specifications, and therefore the validation is more inefficient. Our solution is to translate the boolean attributes as predicates, then we can use the Alloy booleans operators. In this way we can traduce the *query married* as follows:

```
pred Pre_Person_married(s: State,
                        self: s.Persons) {
}

pred Post_Person_married(s: State,
                         self: s.Persons) {
  self.Person_spouse /= none
}

pred Person_married(s: State,
                    self: s.Persons) {
  Pre_Person_married(s, self)
  Post_Person_married(s, self)
}
```

4.4 Commands

A command is translated as a predicate, which has two states as parameters: *s* and *s'*. Informally, *s* represents the system state before the command execution, and *s'* represents the new state after the command execution. For example, the translation for *marry* and *divorce* is:

```
pred Pre_Person_marry(s: State,
                      self: s.Persons,
                      other: s.Persons) {
  self /= other &&
  self.(s.Person_spouse) = none &&
  other.(s.Person_spouse) = none
}

pred Post_Person_marry(s, s': State,
                      self: s.Persons,
                      other: s.Persons) {
  self.(s'.Person_spouse) = other &&
  other.(s'.Person_spouse) = self
}

pred Person_marry(s, s': State,
                      self: s.Persons,
                      other: s.Persons) {
  Pre_Person_marry(s, self, other)
  Post_Person_marry(s, s', self, other)
  self.(s'.Person_cash) =
   self.(s.Person_cash)                  (1)
  other.(s'.Person_cash) =
   other.(s.Person_cash)                 (2)
  Delta(s, s', self + other)             (3)

}

pred Pre_Person_divorce(s: State,
                      self: s.Persons) {
  self.(s.Person_spouse) != none
}
```

```
pred Post_Person_divorce(s, s': State,
                    self: s.Persons) {
  self.(s.Person_spouse) = none &&
  (self.(s.Person_spouse).
   (s'.Person_spouse)) = none
}

pred Person_divorce(s, s': State,
                    self: s.Persons) {
  Pre_Person_divorce(s, self)
  Post_Person_divorce(s, s', self)
  self.(s'.Person_cash) =
   self.(s.Person_cash)
  self.(s.Person_spouse).
  (s'.Person_cash) =
   self.(s.Person_spouse).
   (s.Person_cash)
  Delta(s, s', (self +
              self.(s.Person_spouse)))
}
```

Where the lines (1) and (2) express that attribute *cash* of objects *other* and *self* does not change after executing the command *marry*.
The line (3) invokes the *Delta* predicate, which expresses that the objects not referenced in the postcondition are not changed by the command. The definition of *Delta* is:

```
pred Delta(s, s': State, p0: s.Persons) {
  s'.Persons = s.Persons
  all obj: s.Persons - p0 |
    obj.(s'.Person_spouse) =
     obj.(s.Person_spouse) &&
    obj.(s'.Person_cash) =
     obj.(s'.Person_cash)
}
```

these formulae must be included in the translation since a semantic incompatibility between BON and Alloy. Meanwhile in the first if an attribute is not referenced in the postcondition this means that it is not changed by the command. On the other hand, in Alloy the not referenced attributes may take any value.

4.5 Invariants

One of the interesting features of the translation from BON to Alloy is that it allows us to verify if:

- Commands satisfies invariants.
- The class invariants are satisfiable, that is, they are consistent.

To translate an invariant we use *pred* and *assert* paragraphs, the first ones are used for expressing the invariant as a logical formula. Meanwhile the second ones are used for checking the correctness of commands with respect to invariants. We have to remark that the predicates used for expressing the invariants have as parameters two states.
For example, the invariant of the example given in this section is:

```
pred Person_Inv(s: State) {
  all self: s.Persons |
    self.(s.Person_spouse) =>
     (self =
      self.(s.Person_spouse).
      (s.Person_spouse))
}
```

Where the quantifiers over *s.Person* are needed since the invariant is about instances of *PERSON*. Additionally, we can add an *assert* paragraph for expressing that the invariant is validated by the command. For example, the command *divorce* can be translated as follows:

```
assert {
  all s, s': State |
  all self: s.Persons |
    Person_Inv(s) &&
    Person_divorce(s, s', self) =>
     Person_Inv(s')
}
```

If it is the case that the model has several invariants, suppose:

```
Inv1,..., InvN
```

we must to verify that there exist a model which satisfy all the invariants (i.e., the specification is consistent), we can check it by means of a formulae which combines all the invariants, that is:

```
pred Consistent(s:State) {
  Inv1(s) && Inv2(s) && ... && InvN(s)
}
```

then we can verify automatically using the command: *run Consistent*, in the Alloy Analyzer.

4.6 Verifying Inheritance

Inheritance is one of the most useful tools of object-oriented languages, and therefore it is supported by BON. Furthermore, the concept of inheritance plays an important role in BON:

- The superclass invariants must be satisfied by the subclasses.
- The preconditions in methods could be only weakened by subclasses.
- The postconditions in methods could be only strengthened by the subclasses.
- The type must be preserved by subclasses.

Actually, inheritance is a relation of refinement between contracts. To deal inheritance using automatic tools, BON adds in the subclass methods - using disjunctions - the precondition defined in the superclass. In the same way, postconditions and invariants are added by means of conjunctions. This methodology reflects the status of refinement of inheritance in BON. However, this methodology could have some problems in the practice:

- The invariant in an subclass may contradict the superclass invariant, and therefore the subclass never will satisfy its invariant.
- The postcondition in a routine (in a subclass) could be incompatible with the one of the superclass, and then a exception is always fired when this routine is invoked.
- The precondition in a routine could have the same problem that we described before, doing the precondition impossible to satisfy.

To detect the arising of these problems we can use Alloy, with this purpose in mind we can add the following proof obligations: (*asserts*) During the translation:

- The conjunction between the precondition of a inherited precondition and the former one must be satisfiable.
- The conjunction between the invariants of the superclass and subclass must be consistent.
- The conjunction between the postcondition (the new one and the old one) must be consistent.

On the other hand, we can express the subtyping relation between classes by means of the Alloy keyword: *extends*. For instance:

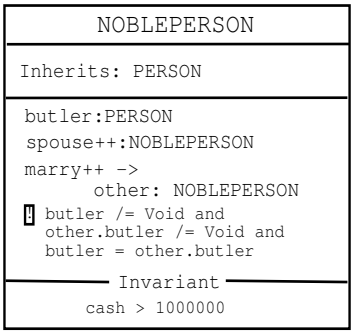


Figure 2: The NoblePerson class

The translation of the types in the model is:

```
sig NoblePerson extends Person {}

sig State {
  Persons: set Person,
  NoblePersons: set NoblePerson,
  Person_cash: Persons -> Int,
  Person_spouse: Persons ->
                 lone Persons,
  NoblePersons_butler: NoblePersons ->
                       lone Persons
} { NoblePersons in Persons }          (4)
```

```
fact {
  all s: State | pn: s.NoblePersons |
    pn.(s.Person_spouse) in
      s.NoblePersons                   (5)
}
```

where; the line (4) guarantees that states preserve the inclusion relation between the *NoblePerson* objects and *Person* objects. Meanwhile in (5) is expressed that *spouse* is redefined by *NoblePerson*.
The *feature marry* of *Person* is traduced in the usual way. In the following example is showed the translation of the *marry* command in the class *NoblePerson*.

```
pred Pre_NoblePerson_marry(s: State,
             self: s.NoblePersons,
             other: s.NoblePersons) {
}

pred Post_NoblePerson_marry(s,
             s': State,
             self: s.NoblePersons,
             other: s.NoblePersons) {
  self.(s'.NoblePerson_butler) !=
   none &&
  other.(s'.NoblePerson_butler) !=
   none &&
  self.(s'.NoblePerson_butler) =
   other.(s'.NoblePerson_butler)
}

pred NoblePerson_marry(s,
             s': State,
             self: s.NoblePersons,
             other: s.PesonaNobles) {
  (Pre_Person_marry(s,
               self, other) ||
   Pre_NoblePerson_marry(s,
              self, other))            (6)
  (Post_Person_marry(s, s',
              self, other) &&
   Post_NoblePerson_marry(s,
         s', self, other))             (7)
  self.(s'.Person_cash) =
   self.(s.Person_cash)
  other.(s'.Person_cash) =
   other.(s.Person_cash)
  Delta(s, s', self + other)
}
```

Note that in line (6) the precondition of *marry* is weakened, and in line (7) the postcondition is strengthened. If we find a model that satisfies this specification then we can ensure that the new preconditions and postconditions are consistent with respect to the superclass assertions. The point is that we can check this using the Alloy Analyzer tool (by means of the command run Noble_marry).
Finally, it is interesting to show in what manner we can verify the consistency of the subclass invariant with respect to the one of the superclass.

To do this we add, by means of a conjunction, the invariant of the parent in the subclass. For example:

```
pred NoblePerson_Inv(s: State) {
  all self: s.NoblePersons |
    self.(s.cash) > 1000000 &&
    Person_Inv(s)
}
```

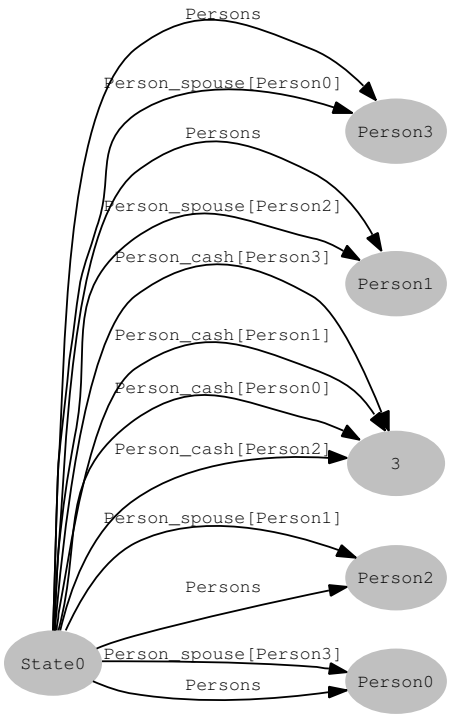
Now we can run the command *run NoblePerson_Inv* using the Alloy Analyzer.
In the next section we describe the tool Bon2Alloy which implements the translation presented in this section.

5. DARWIN TOOL

Darwin Tool is a tool which allows us to edit BON diagrams. Additionally, this software translates, automatically, BON diagrams to Alloy specifications. At this moment, we have only implemented a part of the translation, it includes: classes, client-supplier relations, and the propositional logic of BON.

5.1 An example

We have tested the behavior of the tool with several examples, in this paper we show only one of them since space restrictions.
In figure **??** we showed an example of a BON design. The model was verified using Darwin Tool and the Alloy Analyzer. We have obtained a logical model of the specification, which ensures that it is consistent. The logical model found by Alloy is the following:



We do not show the Alloy specification since space restrictions.

6. CONCLUSIONS AND FURTHER WORK

In this article we have described a method to translate BON design to Alloy specifications. The principal virtue of this method is that it can be done automatically by means of software tools. This, and the feasibility of verifying Alloy specifications, implies that the final user can remove bugs from the designs in an easy way.
On the other side, using this methodology in the BON development process allows us to find errors during the design of a software system, giving more reliability to the other development phases. However, we have a lot of work to do:

- To formalize the translation and to prove its correction.
- To do the BON language more inclusive, including relational operators.
- Support in the Darwin Tool all the BON features.

7. REFERENCES

[1] Kim Waldn and Jean-Marc Nerson. “Seamless Object-Oriented Software Architecture”. Addison-Wesley 1994.

[2] Richard F. Paige. “An Introduction to BON”. August 1999.

[3] Castro Pablo, Baum Gabriel. “Integrando BON con Alloy”. CACIC 2003.

[4] D. Jackson. “Micromodels of Software: Lightweight Modeling and Analysis with Alloy”. February 2002.

[5] D. Jackson. “Alloy 3.0 Reference Manual”. May 2004.

[6] Richard F. Paige and Jonathan S. Ostroff. “A Comparison of Business Object Language and the Unified Modeling Language”. May 1999.

[7] B. Meyer. “Object-oriented software construction”. Prentice-Hall. New Jersey, 1988.

[8] C.A.R. Hoare. “An Axiomatic Basis for Computer Programming,” Communications of the ACM, Oct. 1969.