Guiding Specification and OO implementation of Data Types

Isabel Nunes
Faculty of Sciences
University of Lisbon
Portugal
in@di.fc.ul.pt

Vasco Vasconcelos
Faculty of Sciences
University of Lisbon
Portugal
vv@di.fc.ul.pt

Antónia Lopes
Faculty of Sciences
University of Lisbon
Portugal
mal@di.fc.ul.pt

ABSTRACT

Design by contract (DBC) is among the most popular techniques that are taught in introductory programming courses aiming at helping students to learn how to construct correct and robust software. Although we recognize the important role played by formal design as supported by DBC techniques, we have experienced for several years the frustration of not being able to guide students in writing contracts, both that fully specify all the relevant properties, and are monitorable. The fact that students are left with very poor specifications leads them to perceive contracts as unnecessary and even irrelevant, discouraging the further application of DBC. We addressed these problems through the adoption of property-driven algebraic specifications for the description of the observable behavior of programs. Our approach comprises a tool-assisted refinement process that supports the run-time checking of implementations against specifications. In this paper we present the approach and report on our experience of using it.

Keywords

Algebraic specification, implementation checking, design by contract

1. INTRODUCTION

Teaching object-oriented software development at the University of Lisbon comprises a three-semester programme, according to the objects-first approach of the ACM Computer Science Computing Curricula [17], where students learn basic skills of OO programming, specification, analysis and design, which they will further use in other courses of their BsC. The first course gives students early exposure to the concept of object and class, client and supplier, while introducing more traditional control structures. The second course covers algorithms and fundamental data structures, while the third focus on software engineering topics, including OO development using design patterns.

This programme is based on the methodology of design by contract (DBC) in the sense that students are taught to built responsible client and supplier classes — client classes invoke methods only when their pre-conditions are met, and supplier classes provide results according to method post-conditions.

Difficulties arise in the use of the design by contract methodology, namely with i) contract extension in inheriting classes, ii) increase of class coupling when writing contracts for methods in presence of clientship transitivity, and iii) writing fully monitorable contracts that express all the desirable properties. We identified and proposed solutions for the first two problems [9, 10]. The third one was identified in the evaluation of the aforementioned three-semester course [11]. This paper presents our approach to surpassing this problem, which arises when students start writing and implementing specifications of abstract data types.

The rest of the paper is organized as follows. Section 2 describes the problems that arise when adopting a design by contract approach to data type specification and implementation. It also describes the use of model-based algebraic approaches to specification and points its limitations within the context of a second semester course. Section 3 describes the framework we developed for writing, implementing, and checking property-driven specifications, and that we currently apply in the second course. Section 4 reports on our experience in using the framework. It also describes benefits and limitations of our approach as well as topics that need further work.

2. MOTIVATION

When following the design by contract methodology [14] we experienced the frustration of not being able to guide students in writing contracts, both that fully specify all the relevant properties, and that, at the same time, are monitorable.

Contracts are built from boolean assertions, thus any method invoked within an assertion must return a value. Furthermore, contracts should refer only to the public features of the class because client classes must be able not only to understand contracts, but also to invoke operations that are referred to in them — e.g., clients must be able to test preconditions.

It is important that students test their classes, in particular against contract violations. To be monitorable, a contract cannot have side effects, thus it cannot invoke methods that modify the state.

These restrictions bring severe limitations to the kind of properties we can express directly through contracts. Unless we define a number of, otherwise dispensable, additional methods, we are left with very poor specifications that discourage the further application of DBC. Furthermore, as argued by Barnett and Schulte [3], contract specifications do not allow the level of abstraction to vary and do not support the specification of components independently of the implementation language and its data structures.

```
Java class

public class ArrayStack implements Cloneable {
   public void clear () {... }
   public void push(int i){... }
   public int top (){... }
   public int size (){... }
   public boolean isEmpty (){... }
   ...
}
```

Figure 1: Java implementation of an integer stack.

As an example, let us analyze the support given by DBC to the specification of an integer stack through the integration of assertions in the class `ArrayStack` in Figure 1. Following Meyer [14], and using Java and JML [13] rather than Eiffel, this could be achieved by adding the following assertions.

In method `void clear()` the post-condition
  `ensures size() == 0;`
In method `void push(int i)` the post-condition
  `ensures top() == i && size() == \old(size()+1);`
In method `void pop()` the pre-condition
  `requires !isEmpty();`
and the post-condition
  `ensures size() == \old(size()-1);`
In method `int top()` the pre-condition
  `requires !isEmpty();`
In method `boolean isEmpty()` the post-condition
  `ensures result <==> size() == 0;`

We cannot find suitable post-conditions to express and monitor the property that says that popping a stack right after having pushed an integer, leaves the stack unmodified. The inclusion of a post-condition in method `push` with the flavor of `pop().equals(\old(clone()))` would not work because `pop` is a `void` method. Unless we have methods that allow to inspect the whole structure of the data type elements without modifying it (for instance a method `int element(int i)` for inspecting the *i*-th element of the stack), we are not capable of writing complete monitorable post-conditions. These inspection methods are, in general, artificial, and even against the nature of the type itself and, hence, they are not a solution to the problem. This example shows that, whenever a specification is implemented by a mutable type, there are properties that can not be expressed as monitorable contracts of the class.

Algebraic specification [2, 4, 7] is another well-known approach to the specification of software systems that supports specification at a higher-level of abstraction. Algebraic approaches can be divided into two classes: *model-oriented* and *property-driven* specifications.

Model-oriented approaches to specification, like the ones followed by users of Z [16], Larch [8], JML [13] and AsmL [3], definitely prevail within the OO community. In most of these approaches, the behavior of a class is specified through a very abstract implementation based on primitive, but not necessarily basic, elements available in the adopted specification language. Implementations can be tested against specifications by means of runtime assertion-checking tools, but this requires that an *abstraction function* be explicitly provided. In JML, for instance, a concrete implementation is expected to include JML code defining the relation between concrete and abstract states.

For example, the abstract class `UnboundedStack`, a model-based specification of stacks taken from the JML distribution [1], relies on sequences (more concretely on objects of type `JMLObjectSequence`, a class belonging to the distribution of JML). The following two annotations are part of `UnboundedStack` abstract class:

```
public model JMLObjectSequence theStack;
public initially theStack != null &&
  theStack.isEmpty();
```

The `JMLObjectSequence` class defines immutable sequences, including a series of methods for sequence manipulation from which the methods `trailer()`, `insertFront()`, that are used in this specification, are examples. The model underlying `JMLObjectSequence` is a finite sequence of elements. The post-conditions

```
theStack.equals(\old(theStack.trailer()));
theStack.equals(\old(theStack.insertFront(x)));
```

belong to `UnboundedStack` methods `pop()` and `push(...)`, respectively.

When a specific implementation of `UnboundedStack` is defined, it is necessary to explicitly describe the relation between the `JMLObjectSequence theStack` and the structure that is chosen to store the stack elements. This relation is known as the *abstraction function*. Figure 2 exemplifies the definition of this relation as in [1].

Although we recognize the important role played by model-based approaches in general, we believe that these kind of specifications are difficult to a first grade student to understand and master. They involve additional data types that she has to learn to use. Moreover, the definition of the appropriate abstraction mapping from concrete implementations to the specification can be rather difficult to obtain to most of our students.

3. OUR APPROACH

```
Java class

public class UnboundedStackAsArrayList extends UnboundedStack {

  protected ArrayList elems;
  //@      in theStack;
  //@      maps elems.theList \into theStack;

  /*@ protected represents theStack <- abstractValue();
   @  protected represents_redundantly theStack \such_that
   @     (\forall int i;
   @         0 <= i && i < elems.size();
   @         elems.get(i) == theStack.itemAt(i) );
   @*/

  /*@ protected pure model JMLObjectSequence abstractValue(){
    @   JMLObjectSequence ret = new JMLObjectSequence();
    @   Iterator iter = elems.iterator();
    @   while (iter.hasNext()) {
    @     ret = ret.insertBack(iter.next());
    @   }
    @   return ret;
    @ }
    @*/
…
}
```

Figure 2: Partial view of an implementation of `UnboundedStack`.

Contrasting with the above, *property-driven* specifications [4, 6] can be very simple and concise for certain classes of types, in particular for *Abstract Data Types* (ADTs). In this case, the observable behavior of a program is specified simply in terms of a set of abstract properties. The simplicity and expressive power of property-driven specifications on the one hand, and the necessity for students to test their classes against specifications on the other hand, led us to the development of a framework for creating and testing property-driven specifications against implementations.

In this framework, students proceed as follows.

1. Build algebraic specifications — structured as a *specification module* — to abstractly specify their data types.
2. Input the specifications to a *Specification Analyzer* to obtain feedback on syntax and type correctness.
3. Build Java interfaces from the above specifications; simultaneously, write a *refinement mapping* that relates *i)* each interface with the sort it implements, and *ii)* each interface method with the specification operation it implements.
4. Input the specifications, the refinement mapping, and the interfaces to a *Refinement Mapping Analyzer* to obtain feedback on their coherence.
5. Build classes that implement the above interfaces.
6. Input the specifications, the classes and the refinement mapping to a *Contract Generator* that automatically wraps the provided classes so that, when executed, the behavior of each class is checked against the corresponding specification.

Through the rest of this section we briefly describe significant details about each of the above steps.

Specifications and Modules

The specification language is, to some extent, similar to many existing specification languages. In general terms, it supports the description of partial specifications with conditional axioms. It has, however, some specific features, including the classification of operations in different categories, and strong restrictions on the form of the axioms. Figures 3 and 4 present two examples.

```
specification

import IntegerSpec
sort   IntStack
constructors
   clear: IntStack --> IntStack
   push: IntStack Integer --> IntStack
observers
   top: IntStack -->? Integer
   pop: IntStack -->? IntStack
   size: IntStack --> Integer
derived
   isEmpty: IntStack
domains s: IntStack
   top(s),pop(s): if not isEmpty(s)
axioms s:IntStack, i:Integer
   top(push(_,i)) = i
   pop(push(s,_)) = s
   size(clear(_)) = zero(_)
   size(push(_,s)) = suc(size(s))
   isEmpty(s) if size(s)= zero(_)
   not isEmpty(s) if not (size(s)= zero(_))
```

Figure 3: Specification of integer stacks.

```
specification

sort Integer
constructors
   zero: Integer --> Integer
   suc: Integer  --> Integer
   pred: Integer  --> Integer
observers
   _<_: Integer Integer
axioms i,j:Integer
   zero(_) < suc(zero(_))
   pred(zero(_)) < zero(_)
   pred(i) < j if i<j
   suc(i) < suc(j) if i<j
   pred(i) < pred(j) if i<j
   pred(suc(i)) = i
   suc(pred(i)) = i
```

Figure 4: Specification of integers.

A specification defines exactly *one* sort — the main sort, and the first argument of every operation and predicate in the specification must have that sort. Furthermore, operations are classified as `constructors`, `observers` or `derived`. These categories comprise, respectively, the operations with which all values of the type can be obtained, the operations that provide fundamental information about the values of the type, and the operations where the provided information can be obtained through the remainder operations. Predicates can only be classified as either `observers` or `derived`.

Specifications are partial because operation symbols declared with →? can be interpreted by partial functions. In the section `domains`, we describe the conditions under which interpretations of these operations are required to be defined. For instance, in the specification of integer stacks, both `top` and `pop` are declared as partial operations. They are, however, required to be defined for all non empty stacks.

As usual in property-driven specifications, properties of operations and predicates can be expressed through axioms, which in our case are closed formulæ of first-order logic restricted to some specific forms [15]. Essentially, we have axioms *i)* that relate constructors, *ii)* that define the result of observers on constructors, *iii)* that describe derived operations/predicates results on generic instances of the sort, and *iv)* that pertain to sort equality.

Notice that, because operations may be interpreted by partial functions, a term may not have a value. The equality symbol used in the axioms represents strong equality, that is to say, either both sides are defined and are equal, or both sides are undefined.

Specifications may declare, under the `import` section, references to other specifications, and may use external symbols, i.e., sorts, operations and predicates that are not locally declared. For instance, the specification of integer stacks imports `IntegerSpec` and uses sort `Integer` and operation symbols `zero` and `suc`, which are external symbols. Notice that the specification of integers in Figure 4is self-contained since it does not contains any external symbol. We call it a *closed specification*.

The meaning of external symbols is only fixed when the specification is embedded, as a component, in a *module*. A *module* is simply a surjective function from a set N (of names) to a set of specifications, such that, for every specification: *(i)* the referenced specification names belong to N, and *(ii)* the external symbols are provided by the corresponding specifications in the module. The set N defines the set of components of the module. For instance, by naming the two specifications presented in Figures 3 and 4 as `IntStackSpec` and `IntegerSpec`, respectively, we obtain a module `IntegerStack`.

Students are provided with the following guidelines for building specifications [18]:

1. Identify your sort. Below, let it be `S`.
2. Identify the operations and predicates and define their signatures. The first parameter of each signature should be `S`.
3. Classify the operations according to the three categories.
   - Start with the `constructors`. Candidates are operations with signatures of the form `op: S, ...` →`S`. Do we need them all? For each of them, check whether it produces an intrinsically *new* value of sort `S`. If the resulting value can be obtained otherwise, then the operation is not a constructor.
   - Move to the `observers`. Candidates are all the remaining operations (and predicates). Does each of them allow to observe a *different* facet of the values of the sort? Or can you obtain one of them by using the remaining operators?
   - The remaining operations are `derived`.
4. Check the domain of each operation. Are there operations that are not defined in the whole extension of its domain? Are there particular values that make no sense as parameters to the operation? Record the situations for which the operation is required to be defined in the `domains` section, using expressions of the form `op(X, Y) if ...`. For instance,
   ```
   pop (S) if not isEmpty (S);
   ```
5. Write the axioms.
   - Prepare one axiom for each pair observer-constructor. Exceptions are the pairs ruled-out by domain conditions. For each constructor `c: S T` →`S`, and each observer `o: S U` →`V`, use an axiom whose left hand side is of the form `o (c (X, Y), Z)` where `X` has sort `S`, `Y` has sort `T`, and `Z` has sort `U`. Example:
     ```
     pop (push (S, E)) = S;
     ```
   - For each derived operation `d: S T` →`U` (or predicate `d: S T`) write an axiom whose left hand side is of the form `d(X,Y)`. Example:
     ```
     isEmpty (S) if size (S) = 0;
     ```
   - Add the axioms that pertain directly to sort equality. For example, in a specification for rational numbers, one may want an axiom of the form:
     ```
     F = G if
       num(F)*den(G) = num(G)*den(F);
     ```
   - Prepare the axioms that relate constructors. For example, for sets, we can have:
     ```
     add(add(A, X), Y) = add(add(A, Y), X);
     ```
6. Did not succeed in writing the axioms? Perhaps you need a different classification in step 3, or even a different choice of operations in step 2.

Specification Analyzer

The *Specification Analyzer* tool verifies the syntactic and semantic correctness of a specification module. A specification must conform to rules related to the signatures, domains, axioms, and external names of the specification, as follows.

Signatures are such that

- The first parameter of a signature must have the sort under specification (the *main* sort);
- The result of any constructor operation must have the *main* sort (therefore a predicate cannot be a constructor).

Domains are of the form

$$f(x_1, \ldots, x_n) \quad \textit{if} \quad \phi$$

where f is an operation (not predicate), $x_1, \ldots, n_n$ are variables, and ϕ is a formula.

Axioms must have one of the following four basic structures:

$$f(t_1, \dots, t_n) = t \quad \mathit{if} \quad \phi \qquad (1)$$
$$p(t_1, \dots, t_n) \quad \mathit{if} \quad \phi \qquad (2)$$
$$\mathit{not} \quad p(t_1, \dots, t_n) \quad \mathit{if} \quad \phi \qquad (3)$$
$$x_1 = x_2 \quad \mathit{if} \quad \phi \qquad (4)$$

where f is an operation, p is a predicate, t is a term, x is a variable, and ϕ is a formula. In either case the condition *if* ϕ is optional. The following restrictions apply to these four structures.

- In (1) if f is a *constructor* than t_1 must be either a variable or a constructor applied to variables; the remaining arguments, t_2 to t_n, must all be variables.
- In (1), (2) and (3) if f or p are *observers* than t_1 must be a constructor applied to variables.
- In (1), (2) and (3) if f or p are *derived* than all its arguments, t_1 to t_n, must be variables.
- In (4) x_1 and x_2 must be variables of the *main* sort.

External Names. Any external name, be it a sort which is not the *main* sort, be it an operation that is not defined in the signatures part of the specification, must be specified by some of the imported specification in the same module.

Refinement Mappings

For each specification in a module, we choose a Java type (interface or primitive) for its representation. In this process, we build a *refinement mapping* $\mathcal{R}$ between the module and the collection of Java types. This mapping also describes the correspondence between the operations and predicates in the specifications and the methods in the interfaces. Furthermore, for specifications that are mapped into primitive types, the mapping defines how operations and predicates are expressed in terms of built-in Java operations. Only closed specifications can be implemented by primitive types.

An admissible refinement mapping for the `IntegerStack` module is presented in Figure 5. It says that specification `IntStackSpec` is mapped into type `ArrayStack`, whereas the sort `Integer` is implemented by the Java primitive type `int`.

The following are the guidelines given to students to build refinement mappings. For each class involved (`ArrayStack` in Figure 5):

1. Choose the nature of the interface. Will it describe imperative (mutable) objects, or immutable objects? Does pushing a value into a stack yields a new stack, or, on the contrary, changes the state of the target object?
2. Provide a mapping for each operation in the specification. Ignore the first parameter in each operation. When the resulting sort of an operation is the sort under refinement, choose between an immutable or an imperative method. For example, choose between `Stack push()` and `void push()`.

```
refinement mapping

IntegerSpec is primitive int
  zero(x1:Integer):Integer                is 0
  suc(x1:Integer):Integer                 is x1 + 1
  pred(x1:Integer):Integer                is x1 - 1
  _<_(x1:Integer, x2:Integer)             is x1 < x2

IntStackSpec is type ArrayStack
  clear(s:IntStack):Stack                 is void clear()
  push(s:IntStack,i:Integer):IntStack     is void push(int i)
  pop(s:IntStack):IntStack                is void pop()
  top(s:IntStack):Integer                 is int top()
  size(s:IntStack):Integer                is int size()
  isEmpty(s:IntStack)                     is boolean isEmpty()
```

Figure 5: An example of a refinement mapping.

Notice that a refinement mapping may define a mapping from two different components into the same type (interface or primitive). This is extremely useful since it promotes the writing of generic specifications that can be reused in different situations, as illustrated in reference [15].

Refinement Mapping Analyzer

The *Refinement Mapping Analyzer* tool verifies the syntactic and semantic coherence of a specification module, a refinement mapping $\mathcal{R}$, and a set of interfaces.

Mappings are subject to some constrains:

- Predicates must be bound to methods of type boolean.
- Only sorts defined in closed specifications can be bound to primitive types.
- Every $n+1$-ary operation or predicate $f(s, s_1, \dots, s_n)$ must be bound to an n-ary method $m(t_1, \dots, t_n)$ such that t_i is the type that, according to $\mathcal{R}$, implements sort s_i.

Implementing the Interfaces

For building classes that implement the interfaces, students must follow the following guidelines:

1. Name your class. A good choice is a compound name, where the first part describes the implementation, while the second is the name of the interface. Examples include `IntFraction`, and `ArrayStack`.
2. Choose the representation (the class's attributes); write a body for all the methods in the interface.
3. Write one or more Java constructors. For each one, pick a `constructor` in the specification, and make sure that the Java constructor leaves the object as prescribed by the post-condition of the chosen specification constructor. For example, an obvious choice for a stack Java constructor is to pick the method `clear`, and write:

```
public ArrayStack() { clear(); };
```

Checking Classes Against Specifications

The behavior of classes can be tested against the specifications they are supposed to implement, by using the *Contract Generator* tool. When given a specification module, a refinement mapping, and the classes that are to be tested, the tool creates:

- An immutable version of each original class — these classes are equipped with contracts that result from automatically translating the domains and axioms of the corresponding specifications;
- A wrapper class for each original class — such wrappers redirect every method call to the original class, via the corresponding method in the immutable class.

Whenever a class, client to the original classes, is executed within the context of the tool, every call to a method `m` in any of those classes is monitored (that is, pre and post-conditions of `m` are evaluated). Users are given feedback that identifies the kind of violation: pre or post-condition.

All details, including the architectural model, and the rules for contract generation are described elsewhere [15].

4. EXPERIENCE AND CONCLUSIONS

In 2002, we reported on our experience with the contract-based objects-first approach, encompassing three semesters [11]. Here we concentrate on the second course — Algorithms and Data Structures.

The approach described in this paper has been refined over the years. Students are exposed to specifications to all the data structures studied in the course [18]. They also define less common data structures in their course work, such as a Video Disk (essentially a bonded capacity, index data structure whose elements are of distinct sizes).

The introduction of the *guidelines for building specifications* (Section 3) represented a major leap in the students ability to prepare meaningful and complete specifications. Students not only have a method to attack the problem, but they are also confident that they did not forget any axiom. This confidence is reinforced by the possibility of checking the coherence of their specification, using the *Specification Analyser* tool.

This strongly contrasts with what used to happen before, when students were taught to write contracts (first using iContract [12] and later JML). Students were initially very receptive to the idea of contract monitoring as a technique that helped them to find bugs in their implementations, but soon they realized that many of the important properties of their types were impossible to express through monitorable assertions.

The idea underlying the approach we developed, which encompasses automatic generation of contracts, was to gather the possibility of building meaningful and complete specifications with the capacity of checking implementations.

Using the approach described in this paper, students are able to monitor *all* the axioms in their specifications. One of the remaining problems has to do with some difficulties students now face in interpreting the feedback obtained by monitoring the contracts, since this is given in terms of contracts that they do not see, and that they would scarcely understand even if they did. Students must interpret contract violations in the context of properties they have defined through algebraic specifications.

Further work addresses this difficulty by trying to present contracts in such a way that is, on the one hand, understandable to students, and on the other, closely related to the original specification that they wrote.

Acknowledgments

Thanks are due to José Luiz Fiadeiro, João Abreu, and Luís Reis for insightful discussions.

5. REFERENCES

[1] The java modeling language examples page. http://www.cs.iastate.edu/ leavens/JML/examples.shtml.

[2] E. Astesiano, H.-J. Kreowski, and B. Krieg-Brckner, editors. *Algebraic Foundations of Systems Specification*. IFIP State-of-the-Art Reports. Springer, 1999.

[3] M. Barnett and W. Schulte. Spying on components: A runtime verification technique. In *Proceedings Workshop on Specification and Verification of Component Based Systems — OOPSLA*, 2001.

[4] M. Bidoit and P. Mosses. *CASL User Manual*. Number 2900 in LNCS. Springer, 2004.

[5] Contract based system development. `http://labmol.di.fc.ul.pt/congu/`.

[6] H. Ehrig and G. Mahr, editors. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Springer, 1985.

[7] J. Goguen, J. Thatcher, and E. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In *Current Trends in Programming Methodology*, volume IV: Data Structuring, pages 80–149. Prentice-Hall, 1978.

[8] J. Guttag, J. Horning, S. Garland, K. Jones, A. Modet, and J. Wing. *Larch: Languages and Tools for Formal Specification*. Springer, 1993.

[9] I.Nunes. Design by contract using meta-assertions. *Journal of Object Technology*, 1(3):37–56, 2002.

[10] I.Nunes. Method redefinition - ensuring alternative behaviours. *Information Processing Letters*, 92/6:279–285, 2004.

[11] I.Nunes and V. Vasconcelos. Contract guided system development, 2002. Presented at the ECOOP'02 Sixth Workshop on Pedagogies and Tools for Learning Object-Oriented Concepts.

[12] R. Kramer. iContract — The Java Design by Contract Tool. In *Proceedings of TOOLS USA'98 conference*. IEEE Computer Society Press, 1999.

[13] G. Leavens, K. Rustan, M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in java. In *OOPSLA'00 Companion*, pages 105–106. ACM Press, 2000.

[14] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall PTR, second edition, 1997.

[15] I. Nunes, A. Lopes, V. Vasconcelos, J. Abreu, and L. Reis. Testing implementations of algebraic specifications with design-by-contract tools. DI/FCUL TR 05–22, DI/FCUL, 2005.

[16] J. Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice-Hall, 1992.

[17] The Joint Task Force on Computing Curricula. Computing curricula 2001 (final report). Available at: http://www.acm.org/sigcse/cc2001.

[18] V. Vasconcelos, I. Nunes, and A. Lopes. From abstract data types to contract annotated java interfaces. Lecture Notes, http://labmol.di.fc.ul.pt/congu/, 2006.