Information and Software Technology 82 (2017) 173–176

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

An idiom to represent data types in Alloy

Rohit Gheyi[a,*], Paulo Borba[b], Augusto Sampaio[b], Márcio Ribeiro[c]

[a] Federal University of Campina Grande, Brazil
[b] Federal University of Pernambuco, Brazil
[c] Federal University of Alagoas, Brazil

ARTICLE INFO

Article history:
Received 8 April 2016
Revised 1 October 2016
Accepted 7 November 2016
Available online 9 November 2016

Keywords:
Data types
Alloy

ABSTRACT

Context: It is common to consider Alloy signatures or UML classes as data types that have a canonical fixed interpretation: the elements of the type correspond to terms recursively generated by the type constructors. However, these language constructs resemble data types but, strictly, they are not.

Objective: In this article, we propose an idiom to specify data types in Alloy.

Method: We compare our approach to others in the context of checking data refinement using the Alloy Analyzer tool.

Results: Some previous studies do not include the generation axiom and may perform unsound analysis. Other studies recommend some optimizations to overcome a limitation in the Alloy Analyzer tool.

Conclusion: The problem is not related to the tool but the way data types must be represented in Alloy. This study shows the importance of using automated analyses to test translation between different language constructs.

1. Introduction

Many programming and formal specification languages support the definition of data types with a fixed unique interpretation. Abstract or algebraic types in functional languages (such as Haskell), for instance, have a fixed initial interpretation: the elements of the type correspond to the terms recursively constructed by interpreting type operations as constructors. Similar, Models in ModulaZ and Z schemas [1], among others, have a fixed interpretation defined in terms of the underlying used representation (model) for the type elements. For example, specifying in PVS a record representing an account that has a balance as:

```
Account = [# balance: int #]
```

the elements of account are defined in terms of the elements of its component (integer). For each integer in this language, there is an element of the account type. The number of elements of *Account* is precisely the number of elements of *int*. In both cases, the language semantics relies on implicit axioms that characterize the fixed interpretation.

Contrasting, languages also support the definition of data types with more flexible, non-unique, interpretations [2]. This is the case of Z given sets, UML classes, and OBJ theories. Any set of elements that satisfies the axioms associated to the type definition and operations is a valid interpretation for the data type. The set of recursively constructed terms might be just one among many possible interpretations. The semantics of these language constructs assume no implicit axioms that characterize a fixed interpretation. If desired, developers should explicitly specify them.

Although previous studies [3] relate Alloy signatures to data types, they do not explicitly specify axioms that characterize such interpretation. In fact, similarly to UML classes and OBJ theories, the semantics of Alloy [3] signatures (Section 2) does not assume a fixed interpretation. As a consequence, analyses such as data refinement checking, which only applies to data types having a fixed canonical interpretation, may be unsound in Alloy. We motivate this problem by applying the traditional data refinement analysis to Alloy signatures that do not represent data types in Section 5.

To avoid this problem, we propose an Alloy idiom to encode data types (Section 3). We illustrate how to explicitly specify in Alloy so called generation and canonicalization axioms that are implicit in other languages. We also show how to perform a sound data refinement in Alloy using the Alloy Analyzer (Section 4). Alloy Analyzer may also be useful for testing the relationship of other language constructs.

* Corresponding author.
E-mail addresses: rohit@dsc.ufcg.edu.br (R. Gheyi), phmb@cin.ufpe.br (P. Borba), acas@cin.ufpe.br (A. Sampaio), marcio@ic.ufal.br (M. Ribeiro).

http://dx.doi.org/10.1016/j.infsof.2016.11.003
0950-5849/

2. Alloy

Next we give a brief overview of Alloy 4. An Alloy model or specification is a sequence of *paragraphs* of two kinds: signatures that are used for defining new types, and constraint paragraphs, such as facts and predicates, used to record constraints. Each *signature* comprises a set of objects (atoms), which associate with other objects by relations declared in the signatures. A signature paragraph introduces a type and a collection of *relations*, called fields, along with their types and other constraints on their included values.

Next we show the Alloy models of a computer memory [3]. An abstract memory (*AbsMemory*) declares a ternary relation (*data*) relating each address to a data item. A concrete memory (*ConMemory*) describes a cache system in terms of the *main* and *cache* ternary relations. The following signatures represent these memories.

```
sig AbsMemory {
  data: Addr -> Data
}
sig ConMemory {
  main: Addr -> Data,
  cache: Addr -> Data
}
sig Addr, Data {}
```

In Alloy, we can declare *facts*, which introduce constraints that always hold. For example, the following fact (*CanonicalizationAxiomAbsMemory*) states that there are no distinct abstract memories with the same data. The *all* keyword denotes the universal quantifier, whereas the => operator denotes logical implication.

```
fact CanonicalizationAxiomAbsMemory {
  all a1,a2:AbsMemory | a1.data = a2.data => a1 = a2
}
```

Predicates are used to package reusable formulae. For example, the abstract and concrete memories [3] define the write operation, as declared next. The concrete operation always writes in the cache. The ++ and -> operators represent the relational override and product, respectively. Notice that both operations can always be applied. There is no precondition.

```
pred absWrite[a,a':AbsMemory, ad:Addr, d:Data] {
  a'.data = a.data ++ (ad->d)
}
pred conWrite[c,c':ConMemory, ad:Addr, d:Data] {
  c'.main = c.main
  c'.cache = c.cache ++ (ad->d)
}
```

Besides predicates and signatures, we can declare *functions*. An abstraction function is provided [3] expressing that the abstract memory associated with a cache system is obtained by taking the contents of the main memory, and overriding them with the contents of the cache, which contains the recent updates. Next we specify an Alloy function representing the retrieve.

```
fun retrieve[c:ConMemory]:AbsMemory {
  {a:AbsMemory | a.data = c.main ++ c.cache}
}
```

Assertions are another kind of constraint paragraph, which declares a set of questions about a model. Suppose we would like to know whether distinct concrete memories have distinct data in the main and cache. The following assertion captures this intention.

Table 1
Summary of alloy keywords and operators.

Keyword/operator	Meaning
all, some	∀, ∃
or, and	∨, ∧
+, !	∪, ¬
=>, ->	⇒, product
++	Override expression
.	Relational join

```
assert differentData {
  all c1,c2: ConMemory |
    c1 != c2 =>
      (c1.main != c2.main) or (c1.cache != c2.cache)
}
check differentData for 3
```

The *!* operator denotes negation. Alloy has some other paragraphs that are used for performing analysis using the Alloy Analyzer tool [5]. Alloy Analyzer can be used to verify whether some property holds for a pre-defined *scope*, which defines the maximum number of objects allowed for each signature during analysis. The simulations performed by the Alloy Analyzer tool are *sound* and *complete up to a given scope*. If it does not yield a counterexample, we cannot conclude that the formulae declared in the assertion are valid since the tool is not a theorem prover. We can gain greater confidence by increasing the scope. Checking the previous assertion with at most three objects for each signature yields a counterexample in which two distinct concrete memories have exactly the same data in the main and in the cache. So, the previous assertion is not deducible from the constraints in the model. We can specify a similar canonicalization fact for concrete memories to avoid the previous counterexample. Table 1 specifies the meaning of all Alloy keywords and operators used in this article.

3. Idiom

An abstract data type comprises a collection of variables, and a list of operations that may change their values [1,2]. Next we propose an idiom to represent a data type *D* in Alloy. First, we create the signature *D* containing a number of relations *ri* that represent data type variables where *Ti* can be a signature or a product of signatures, as described next.

```
sig D {
  r1: T1,
  r2: T2, ...
}
```

For each Alloy signature representing a data type *D*, we must include the generation axiom [1,2]. Next we present a template of a formula declared in a fact representing a generation axiom for *D*. We must quantify over all relations declared in *D*.

```
fact GenerationAxiomD {
  all r1:T1, r2:T2, ... |
    some d:D | d.r1 = r1 and d.r2 = r2 and ...
}
```

In some domains, we may add a canonicalization axiom for each signature representing a data type in a fact. The following fact states that there are no two different objects of *D* with the same values for all relations.

```
fact CanonicalizationAxiomD {
  all x,y: D |
    x.r1 = y.r1 and x.r2 = y.r2 and ... => x = y
}
```

Finally, each data type may include a number of operations. They can be declared in Alloy using predicates. Each predicate must declare at least two parameters representing the state of the data type before and after the operation, as declared next.

```
pred op [d, d': D, ...] {
  ...
}
```

4. Example

Suppose we would like to check in Section 2 whether *ConMemory* refines *AbsMemory* with respect to the *write* operation using the functional data refinement rule [1,6] that was proposed for data types [6]. First of all, we have to check whether *AbsMemory* and *ConMemory* are data types. The example presented in Section 2 does not follow our idiom (Section 3). It does not include the generation axioms for each data type. Next we include them.

```
fact GenerationAxiomAbsMemory {
  all data':Addr->Data |
    some m:AbsMemory | m.data = data'
}
fact GenerationAxiomConMemory {
  all main', cache': Addr->Data |
    some m:ConMemory |
      m.cache = cache' and m.main = main'
}
```

Now we declare a functional data refinement rule [1] in an assertion.

```
assert generalRefRule {
  all a:AbsMemory, c,c':ConMemory, ad:Addr, d:Data |
    a = retrieve[c] and conWrite[c,c',ad,d] =>
      some a':AbsMemory |
        absWrite[a,a',ad,d] and a' = retrieve[c']
}
check generalRefRule for 16 but 2 Addr, 2 Data
```

Performing analysis on the *generalRefRule* assertion in Alloy Analyzer, which supports analyzing high-order quantifications [7], does not yield a counterexample using a scope of at most 16 memories and 2 *Data* and 2 *Addr*. We improve confidence that the data refinement is sound. It is important to mention that the scope given to *AbsMemory* and *ConMemory* must be derived from the scope given to *Data* and *Addr* when performing analysis in Alloy Analyzer. For example, for 2 addresses and 2 data, each memory has 16 values.

5. Related work

Jackson [3, p. 216] states that the concrete memory refines the abstract one presented in Section 2 without the generation axioms. It uses a slightly different assertion than *generalRefRule* to ensure that *conWrite* refines *absWrite*, as described next.

```
assert refinement {
  all c,c':ConMemory, ad:Addr, d:Data, a,a':AbsMemory |
    conWrite[c,c',ad,d] and
    a = retrieve[c] and
    a' = retrieve[c'] => absWrite[a,a',ad,d]
}
check refinement for 10
```

However, checking the data refinement rule (*generalRefRule*) [1] in the specification in Section 2 yields a counterexample. It is not a data refinement. The counterexample generated shows we have to be careful when applying the traditional technique of data refinement to language constructions that resemble data types but, strictly, are not.

Bolton [4] encodes a metamodel of data types in Alloy. Bolton specifies data types using signatures, and represents all possible states as signatures. Bolton explicitly encodes all state transitions of a data type in Alloy. Bolton's idiom may be time consuming to encode when there are a number of states and state transitions. We do not represent a state in a signature. We use predicates to specify state transitions. Malik et al. [8] present a formal translation of a Z subset to Alloy to make analysis and visualization possible for Z users. Malik et al. mention that Z schemas cannot be directly translated to Alloy signatures. We formalize an idiom to represent data types using Alloy signatures.

Estler and Wehrheim [9] translate Z specifications in Alloy to use Alloy Analyzer to check refactorings. It relates a Z data type to an Alloy signature. According to Estler and Wehrheim, the Alloy existential quantifier produces problems within the translation and, even worse, prohibits a simple verification of refinements [9]. Estler and Wehrheim recommend using a total bijective representation relation between the original specifications to overcome this problem in the tool. Ramananandro [10] translates a Mondex case study written in Z to Alloy. It uses Alloy Analyzer to check data refinement. It also relates a Z data type to an Alloy signature. Ramananandro introduces predicates to verify whether the specification had enough properties to define the quantified object to overcome the translation issue. We show this is not a problem related to existential quantifier in Alloy Analyzer. This happens since the considered Z schemas are data types. We have to include the generation axiom.

6. Conclusions

We propose an idiom to specify data types in Alloy. This work clarifies issues found in Alloy Analyzer reported by previous studies [3,9,10]. The problem is not related to existential quantifier in Alloy Analyzer but the way data types are represented in Alloy. The generation axiom may cause state explosion when performing analysis on Alloy Analyzer. Developers may perform some optimizations to overcome this issue, but they must be careful to avoid making unsound analysis. This study shows the importance of using automated analyses to test translation between different language constructs. Automatic analysis tools are far more ruthless than human reviewers [3]. Finally, we recommend encoding part of the translation of other language constructs in Alloy and using Alloy Analyzer to help to avoid similar misunderstandings.

Acknowledgments

We would like to thank the anonymous reviewers. This work was partially supported by CNPq and CAPES grants.

References

[1] J. Woodcock, J. Davies, Using Z: Specification, Refinement, and Proof, Prentice Hall, 1996.

[2] C. Jones, Systematic Software Development using VDM, second ed., Prentice Hall, 1990.
[3] D. Jackson, Software Abstractions: Logic, Language and Analysis, first ed., MIT Press, 2006.
[4] C. Bolton, Using the Alloy analyzer to verify data refinement in Z, ENTCS 137 (2005) 23–44.
[5] D. Jackson, I. Schechter, I. Shlyakhter, Alcoa: the Alloy constraint analyzer, in: ICSE, 2000, pp. 730–733.
[6] C. Hoare, Proof of correctness of data representations, Acta Inf. 1 (4) (1972) 271–281.
[7] A. Milicevic, J.P. Near, E. Kang, D. Jackson, Alloy*: a general-purpose higher-order relational constraint solver, in: ICSE, 2015, pp. 609–619.
[8] P. Malik, L. Groves, C. Lenihan, Translating Z to Alloy, in: Abstract State Machines, Alloy, B and Z, vol. 5977, 2010, pp. 377–390.
[9] H.-C. Estler, H. Wehrheim, Alloy as a refactoring checker? ENTCS 214 (2008) 331–357.
[10] T. Ramananandro, Mondex, an electronic purse: specification and refinement checks with the alloy model-finding method, Formal Aspects Comput. 20 (1) (2007) 21–39.