Software Analysis and Verification — Miniproject

A SAT-based bug finder

June 29, 2007

Lucian Variu

Professor: Viktor Kuncak

1 Introduction

ALLOY is a language for describing structural properties. The declaration syntax is compatible with graphical object models, also it is powerful enough to express complex constraints and is still suitable for fully automatic semantic analysis. The ALLOY analyzer is a tool for analyzing models written in the ALLOY language. The tool generates instances of invariants, it simulates the execution of operations, and checks user-specified properties of models described in the ALLOY language. Ideally the language would be decidable, however the most elementary calculus involving relations is undecidable. The compromise that the ALLOY analyzer makes is that the analysis is sound but not complete. The analysis is guaranteed to be sound because the model returned is indeed a model. On the other hand, if the analysis does not return a model, one cannot conclude that there exists no model. The analysis only considers all potential models up to a given size, specified by a scope that limits the number of atoms in each primitive type. ALLOY and its analyzer are used mainly to explore abstract software designs.

JAHOB is a static analysis and verification system that allows modular analysis of imperative computer programs that manipulate relations as state components. The input language of JAHOB is a subset of Java that has been extended with annotations written as special comments.

The goal of this project is to create a SAT-based bug finder. This goal can be achieved by creating a translation of proof obligations generated by JAHOB to ALLOY models. When the models are obtained the ALLOY analyzer is invoked with a user defined scope and an analysis is performed.

2 A translation example

In this section we give an example of translation from JAHOB proof obligations to corresponding ALLOY models. We shall give a simple banking example. The example is in Java and the code consists of only one class with one method. The class has one variable that represents the account balance and a method that deposits a given amount in the account balance. The annotations require that the amount is greater than zero before the method invocation and ensure that the resulting balance is the sum of the old value of the balance and 1. The Java code, annotated with JAHOB's special comments is given below:

```
class Account {
    private static int checkingBalance;
```

```
    private static void checkingDeposit(int amount)
    /*:
      requires "amount > 0"
      modifies checkingBalance
      ensures "checkingBalance = old checkingBalance + 1"
     */
    {
        checkingBalance = checkingBalance + amount;
    }
}
```

For simplicity we will show here how the proof obligations of the above code that JAHOB prints as debug messages are translated to the corresponding ALLOY model. We can see that the proof obligation consists of the procedure precondition – that requires the amount variable to be larger than zero – that implies the procedure postcondition – that ensures that the amount added to the account balance is equal to one plus the account balance. Of course, these assumptions are false and the analysis of the corresponding ALLOY model should return a counter-example. The proof obligation generated by JAHOB for the above code is the following:

```
Proof obligation:
(comment ''ProcedurePrecondition''
   (intless (0 :: int) (amount :: int))
==>
 comment ''ProcedureEndPostcondition''
 (((intplus
  (Account_checkingBalance :: int) (amount :: int)) :: int)
 =
 ((intplus
  (Account_checkingBalance :: int) (1 :: int)) :: int)))
```

The corresponding ALLOY model for this proof obligation is going to contain an assertion that implements the formula described by the proof obligation above. The model also contains the command for checking the assertion for a given scope and a given bit-length representation for integers. Understanding the model is straightforward due to the intuitive ALLOY language. Our ALLOY model is given below:

```
assert checkingDeposit {
  all amount : Int |
  all Account_checkingBalance : Int |
```

```
  all amount : Int |
  int amount > 0
     implies
  int Account_checkingBalance + int amount
  =
  int Account_checkingBalance + 1
}
check checkingDeposit for 10, 5 int
```

Running the Alloy analyzer on this model reveals a counterexample. This counterexample is due to the way integers are implemented in Alloy. We give a detailed explanation of this behavior in the encountered problems section.

3 Translating Jahobto Alloy

3.1 Translating manually

Translation of Jahob proof obligations to Alloy models may seem an easy task at first. However, finding a completely mechanical way of translation is not that simple. We started out by manually analyzing how Jahob proof obligations should be translated to Alloy models. Using the example above we found that for the case of programs that use only simple integer manipulation we can create the Alloy assertion formula by simply parsing the Jahob proof obligation and converting from Jahob syntax to Alloy syntax. This means that the proof obligation's precondition:

```
(comment ''ProcedurePrecondition''
   (intless (0 :: int) (amount :: int))
==>
 comment ''ProcedureEndPostcondition''
 (((intplus
  (Account_checkingBalance :: int) (amount :: int)) :: int)
 =
 ((intplus
  (Account_checkingBalance :: int) (1 :: int)) :: int)))
```

when written in Alloy syntax is simply:

```
  int amount > 0
     implies
  int Account_checkingBalance + int amount
```

```
=
int Account_checkingBalance + 1
```

By performing this simple syntax translation we only get part of the full ALLOY model we seek. When doing the translation by hand, completing the model is quite easy. One only has to write the assertion header, declare all variables used in the formula, and write the assertion checking statement for a given scope and integer bit-length. Upon completing these steps one can put the model in a file or paste it into ALLOY's GUI and run the analysis. The result of the analysis can return a counter-example in the case an intended property is violated. In our running example this is precisely what happens, because logically $amount > 0 \nRightarrow balance + amount = balance + 1$.

When dealing with more complicated proof obligations generated by JAHOB even the manual translation is not that obvious. The most difficult problem is to find how to define the appropriate ALLOY signatures for the non-integer variables, then translate all the operations regarding these variables to relations using sets, and also track types of all variables.

3.2 Translating mechanically

The problems that one faces when trying to implement a mechanical translation from JAHOB proof obligations to ALLOY models are more numerous and complex then when trying to translate manually. First of all we cannot translate the proof obligation string. We must mechanically manipulate JAHOB's data-structures. The translation code will use JAHOB's term-representation of formulas and convert this representation to ALLOY-like syntax.

Our code first creates a list of all variables and determines their types. Having this information we can already generate the equivalent ALLOY declarations for variables. Next we invoke a function that recursively passes on the term-representation of the proof obligation and outputs the equivalent ALLOY notation for the formula.

The process described so far gives us the core of the translation. The only thing left in order to make our ALLOY model complete is to add the assertion declaration around the body of the model that we produced so far, add the assertion checking statement, and specify the scope of the analysis. At the end of this process we obtain a complete ALLOY model that we write to a file. Our module next invokes the ALLOY analyzer with the newly generated model and reads the result of the analysis. We generate models for each proof obligation generated by JAHOB, the result of the decision process is a conjunction of the results for each of the models that are checked.

3.3 Problems encountered

Coming back to the translation of our running example the model that we generated interestingly reveals a counterexample. This behavior is caused by keeping the bit-length representation of integers constant and increasing the scope in which we search for models. In particular, if the original `balance` has a value near the maximum positive integer then adding a positive `amount` will cause `balance + amount` to wrap around and become a negative integer value. Thus depositing money into the account can make the `balance` smaller.

This behavior is due to the nature of integers in Alloy. In Alloy the integers are represented using a fixed number of bits. Thus if an integer were represented using 4 bits, the possible values were in the range [-8, ..., 7]. The easiest resolution of this problem is to increase the bit-length for representing integers while keeping the maximal scope for the model search.

4 Results

Our results consist of an OCaml module that can translate integer arithmetics based proof obligations generated by Jahob to their corresponding Alloy models. The translation is obtained by recursing Jahob's abstract syntax tree and simplifying it to the extent that it matches Alloy's features. This Ocaml module is integrated in Jahob and it permits the use of Alloy as a decision procedure for the generated proof obligations. The current implementation offers Jahob's user the option of specifying the scope bounding Alloy's model search and also the number of bits to be used in representing integers in Alloy. The module also incorporates some preliminary code for translation of set operations, however translation of set operations is currently not supported.

5 Future work

Our module should be further extended such that is can handle translations of set operations. This can be done by adding to the existing code functions that infer and create Alloy signatures for representing types of set variables in the Jahob proof obligations and then keeping track of the types of each of the set variables.

In order to have a complete translation of Jahob proof obligations to Alloy models further work must find how to translate the remaining components of Jahob's data-type to the equivalent Alloy constructs. When

this is done, the translation will simply follow the idea behind integer arithmetics handling. JAHOB's abstract syntax tree is recursively passed and simplified the semantics matches the features supported by ALLOY.