Recursion and Iteration Support in USE Validator with AnATLyzer

Jesús Sánchez Cuadrado

Modelling and Software Engineering Research Group (http://www.miso.es)
Universidad Autónoma de Madrid (Spain)

Abstract. Model finders enable numerous verification approaches based on searching the existence of models satisfying certain properties of interest. One of such approaches is anATLyzer, a static analysis tool for ATL transformations, which relies on USE Validator to provide fine grained analysis based on finding witness models that satisfy the OCL path conditions associated to particular errors. However it is limited by the fact that USE Validator does not include built-in support for analysing recursive operations and the *iterate* collection operator.
This paper reports our approach to allow USE Validator to analyse OCL path conditions containing recursive operations and *iterate*, with the aim of widening the amount of actual transformations that can be processed by anATLyzer. We present our approach, based on unfolding recursion into a finite number of steps, and we discuss how to take into account practical aspects such as inheritance and details about the implementation.

Keywords: OCL, ATL, USE Validator, Recursion, Iteration, Model finder, Constraint Solver

1 Introduction

Model finders are an important element of many automated verification approaches in the MDE setting, since they are able to find models satisfying certain properties of interest. Concrete examples of such finders are USE Validator [4] and EMFtoCSP [3], which take as input a meta-model and a set of OCL invariants and return a model satisfying the invariants, if any, within a certain scope (e.g., the maximum number of instantiations of the meta-model classes and ranges of attribute values for infinite types such as integers).

As part of our work in the static analysis of ATL transformations [2], implemented in anATLyzer[1], we have used the model finder implemented by USE Validator [4] to enable the precise analysis of certain error types. This analysis involves creating an OCL path condition which is fed into USE Validator to obtain a model that satisfies it, in order to confirm the error if the model can

[1] http://www.miso.es/tools/anATLyzer.html

be found or to discard the error if not. However, USE Validator does not support recursive operations nor the *iterate* collection operation, hence limiting the applicability of our method in some cases.

This paper reports our approach to enable the analysis of recursive operations and expressions containing the *iterate* collection operation in an OCL-based model finder without support for them, focussing on USE Validator. For recursive operations we unfold the recursion upto a number of levels. In the case of *iterate* we similarly convert each call to a sequence of operations that implements a limited number of iteration steps. We have tested our approach with USE Validator but it could be easily implemented for other systems.

Paper organization. Section 2 introduces the context of this work using an example, Section 3 describes the unfolding of direct recursive operations, whereas Section 4 explains the adaptation of the previous procedure for *iterate*. Finally, Section 5 discusses some issues of our approach and concludes.

2 Context and motivation

The context of this work is our static analysis tool for ATL transformations, called anATLyzer. It consists of a type checking phase in which confirmed failures and potential errors are identified. Then, for each potential error, we compute its OCL path condition, which is an OCL constraint that must be satisfied by any source model that would trigger the error at runtime. Afterwards, such path condition is fed into USE Validator to search for a model, a *witness model*, that satisfies the condition. If found, the error is confirmed, otherwise it is discarded. Hence, a key element for this approach to be practical is to maximise the number of path conditions that can effectively be processed by USE Validator. More details about the approach are described in [2].

As an example let us consider a modified excerpt of the CPL2SPL transformation available in the ATL Zoo[2], which establishes a translation between two telephony DSLs. Figure 1 shows an excerpt of the CPL source meta-model, and an examplary listing[3]. This piece of transformation maps every SubAction source element to a LocalFunctionDeclaration in the target, and each Proxy which satisfies the isSimple predicate into a ReturnStat. In ATL, the relationships between rules are established via *bindings*, denoted by ←, which work as follows. The source elements obtained by evaluating the right part of a binding are looked up in the transformation trace, in order to obtain the corresponding target element created by some rule. In the example, the binding in line 20 is evaluated by executing the expression s.contents.statement which retrieves a Node source element. If such source element has been transformed by some rule, the corresponding target element is assigned to the statement feature.

A smell that the transformation behaviour is not as expected is that a source element appearing in the right part of a binding has not been transformed by

[2] `http://www.eclipse.org/atl/atlTransformations/#CPL2SPL`

[3] We added a filter to the SubAction2Function rule and removed a related rule to make the example more illustrative.

any rule. In this setting, anATLyzer features a rule analysis component which is able to analyse rule–binding relationships to determine if a binding is fully covered by all the resolving rules. To analyse the binding in line 20 anATLyzer builds the OCL path condition shown at the bottom of Figure 1, which states the properties that a model for which the binding would be unresolved must satisfy. This OCL invariant is fed into USE Validator to search for a witness model that confirms the existence of the problem.

However, in practice anATLyzer could not perform this particular analysis due to USE Validator not supporting recursive operations, as is the case of Location.statement. Next section describes how anATLyzer unfolds recursion to enable this analysis, while Section 4 explains how we deal with *iterate*.

3 Direct recursion

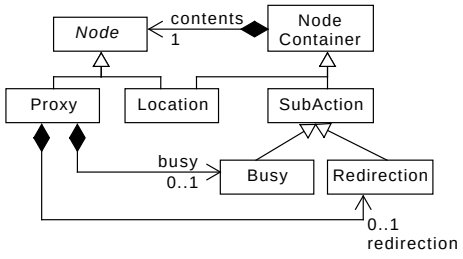
This is the basic recursive schema, in which an operation calls itself in one or more call sites within the operation body. Any OCL specification with an operation featuring even this simple form of recursion is rejected by USE Validator. For the example path condition USE does not try to evaluate the call to Location.statement because it cannot be determined if the operation terminates. Hence, the analysis cannot be carried out.

Our approach to deal with this issue is based on unfolding the recursive operation upto a finite number of steps. We perform the unfolding by copying the original operation n times, so that there are $n+1$ versions of the operation. Then, each version of the operation is rewritten so that the recursive call sites do not invoke the original operation, but the next copy of the operation. The last operation in the sequence just returns OclUndefined.

Listing 1 shows a sketch of this procedure. It takes the desired number of unfoldings (N) and the piece of abstract syntax corresponding to the recursive operation (OP). There are two helper functions, callSites which returns the set of recursive call sites (i.e., a set of *OperationCall* abstract syntax elements that invoke Op) and copy which returns a deep copy of the given abstract syntax element.

```
1  N = Number of unfoldings
2  OP = Original operation
3
4  OP_0 = OP
5  for i = 1 to N
6    CS_{i-1} = callSites(OP_{i-1})
7    foreach cs in CS_{i-1}
8      cs.operationName = OP.operationName + "_" + i
9    end
10
11   OP_i = copy(OP)
12   OP_i.operationName = OP.operationName + "_" + i
13 end
14
15 OP_N.body = OclUndefined
```

Listing 1: Sketch of the unfolding algorithm.




```
1  -- We consider nodes are statements, by default.
2  helper context CPL!Node def: statement : CPL!Node =
3    self;
4
5  -- The "location" node is not a statement.
6  helper context CPL!Location def: statement : CPL!Node =
7    self.contents.statement;
8
9  helper context CPL!Proxy def: isSimple : Boolean =
10   self.busy.oclIsUndefined() and
11   self.redirection.oclIsUndefined();
12
13 rule SubAction2Function {
14   from s : CPL!SubAction (
15           s.contents.oclIsKindOf(CPL!Location) )
16   to t : SPL!LocalFunctionDeclaration (
17     name <- s.id,
18     returnType <- rt,
19     -- Is this binding fully covered by resolving rules?
20     statements <- s.contents.statement
21   )
22 }
23
24 rule Proxy2Return {
25   from s : CPL!Proxy ( s.isSimple )
26   to t : SPL!ReturnStat (
27     ...
28   )
29 }
```

```
SubAction.allInstances()->
  select(s | s.contents.oclIsKindOf(Location))->
  exists(s |
    let _problem_ = s.contents.statement in
      not _problem_.isUndefined() and
      not (if _problem_.oclIsKindOf(Proxy) then
              let s2 = _problem_.oclAsType(Proxy)
               in s2.isSimple()
            else
              false
            endif))
```

Fig. 1: Excerpt of the CPL meta-model (left), two simplified rules of the CPL2SPL transformation (right), and excerpt of the path condition for the problem in line 20 (bottom)

In practice, this procedure needs to be extended to deal with inheritance. This means that it is not enough to duplicate the recursive operation, but every operation that could polymorphically be invoked needs to be duplicated as well.

Listing 2 shows the final result as it is generated to be processed by USE, and complements the OCL path condition presented in Figure 1. Hence, using this method ANATLYZER is able to obtain the witness model shown in Figure 2 that confirms the existence of the problem. As can be seen the model contains the elements required to trigger the problem: SubAction and Location objects to trigger

the SubAction2Function, and a Proxy element which does not satisfy the isSimple predicate, and it is thus not handled by any rule. Since the Proxy element is linked to SubAction via the contents reference, the binding in line 20 will be unresolved. This model has succesfully been obtained because just two unfolding steps are enough in this case. We heuristically set the maximum number of unfoldings to five, but we still do not have any automated mechanism to set parameter to a safe value for those specific cases in which such reasoning could be possible. For instance, the upper bound of a recursive operation (possibly polymorphic) with no parameters would be the maximum number of instances of the class, and the involved subclasses, set as the the model finder scope.

```
abstract class Node
operations
  statement() : Node = self
  statement_1() : Node = self
  statement_2() : Node = self
  statement_3() : Node = self
end

class Location < Node, NodeContainer
attributes
  url : String
  clear : String
operations
  statement() : Node = self.contents.statement_1()
  statement_1() : Node = self.contents.statement_2()
  statement_2() : Node = self.contents.statement_3()
  statement_3() : Node = OclUndefined
end
```

Listing 2: Unfolded code as generated for USE Validator

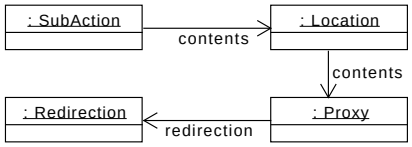


Fig. 2: Witness model obtained for the path condition.

An alternative to this approach is to inline the operation body n times, using *let* expressions to bind parameters. However, we prefer the one presented here because it is easier to handle polymorphic calls as explained.

4 Iterate

The OCL *iterate* collection operation is a general iteration operation with the form col→iterate(it; acc = <init> | <body>). Operationally, it iterates over the elements of the collection assigning them to the it variable in each iteration step.

Each time, the given body is evaluated and the acc variable is updated with the result of the evaluation, so that it has a new value in the following iteration or it is the final result of the operation. As an example, the following code implements the select collection operation in terms of iterate.

```
-- Given: col->select(it | <body>) where col is a Set

col->iterate(it; acc = Set { } |
  if <body> then
    acc->including(it)
  else
    acc
  endif)
```

In practice, USE Validator is able to evaluate most OCL iteration constructs, such as *select*, *any*, etc. However, it cannot evaluate *iterate* which poses a limitation for ANATLYZER since path conditions containing *iterate* cannot be processed.

Our approach to deal with this issue is based on unfolding the iteration steps. Until now we support iteration over sets, applying the following strategy. For each call to *iterate* we generate n operations, being n the number of unfoldings, and each of these operations follows the schema shown in Listing 3. First, we check if the collection is empty (line 10) in case the iteration must terminate returning the currently computed value (acc). If the set is not empty, one element is picked using any (line 13), and then a new set is obtained filtering out the picked element from the original set (line 14). To the best of our knowledge this is the only strategy to implement iteration over sets in OCL. Aftewards, the body of the original iterate is evaluated, and the next iteration operation is invoked. Finally, the recursion is ended at depth n by just returning OclUndefined (line 20).

```
-- Given an expression: col->iterate(it : T_it; acc : T_acc = <init> | <body)
-- where:
--     T_col: the type of the elements of the collection
--     T_it: the type of the iteration variable, which must be compatible with T_col
--     T_acc: the type of the accumulator variable

class ThisModule
operations
  def iterate_aux_i(col : Set(T_col), acc : T_acc) : T_acc =
    if col->isEmpty() then
      acc
    else
      let it : T_it = col->any(_ | true) in
      let rest : Set(T_col) = col->select(v | v <> it) in
      let value : T_col = <body>
       in iterate_aux_{i+1}(rest, value)
    endif
  ...

  def iterate_aux_n(col : Set(T_col), acc : T_acc) : T_acc = OclUndefined

end
```

Listing 3: Schema for unfolding iterate, using USE syntax

We make use of a special class named ThisModule to allow global operations to be defined. Notably, the iteration operations are defined within this class. In this way, every call to iterate is rewritten to an expression similar to thisMod-

ule.iterate_aux0(col, <init>), where the thisModule variable is an instance of ThisModule that must be introduced in the scope of the rewritten expression. We also generate unique identifiers for the iteration operations, to avoid name clashes if there are several calls *iterate* in the same path condition. Finally, we also keep track of the variables in the scope of the original iterate, and if needed, we extend the signature of the iteration operation to pass such variables as parameters.

5 Conclusions and discussion

In this paper we have presented our approach to enable USE Validator analyse recursive operations and the *iterate* collection operation in the context of anATLyzer. In both cases we perform an unfolding of the body of the operations upto a finite number of steps. We have run a small number of tests in which these approaches have shown to be useful, since most of the times a small scope is enough to find the required witness model [1]. Nevertheless, it is part of our future work to carry out more experiments to determine the precision of our approach. In addition, there are some practical considerations to be taken into account, which are discussed in the following.

Given that there is a limit in the number of unfoldings, the last step of the unfolding needs to return some value. Ideally, a *bottom* value should be used to indicate a kind of "stack overflow", in the sense that the recursion has ended prematurely before finishing the computation. In OCL the closest relative to a bottom value is *invalid* which conforms to *OclInvalid*, which in turn conforms to any other type, but any call applied to its unique instance results in invalid itself. However, this is not supported by USE, and thus another value must be used. Selecting such value is difficult in the general case, since it could interact with other expressions processing the return value. We use OclUndefined both for recursion and *iterate* but we are aware that it may affect the accuracy of the solving process.

In this line, an important consequence of unfolding a limited number of times is that the analysis of anATLyzer may not be accurate. A potential error can be wrongly marked as "discarded" only because more unfoldings steps would be needed to provide an accurate answer.

Another issue that affects the accuracy of the approach is that USE only supports sets. Therefore, operations such as *at*, for sequences, cannot be processed. Devising mechanisms to deal with sequences is part of our future work. Hence we aim at studying and adapting other works dealing with these issues, notably the approach proposed in [5] which relies on SMT solving and a more sophisticated unfolding algorithm.

Finally, we have not addressed yet how to unfold mutual recursion, although we believe that a similar strategy is possible, but taking into account the complete call graph of the transformation. This is also part of our future work.

Acknowledgements. This work has been supported by the Spanish MINECO (TIN2011-24139 and TIN2014-52129-R), the R&D programme of the Madrid Region (S2013/ICE-3006), and the EU commission (FP7-ICT-2013-10, #611125).

References

1. A. Andoni, D. Daniliuc, and S. Khurshid. Evaluating the "small scope hypothesis". Technical Report MIT-LCS-TR-921, MIT CSAIL, 2003.
2. J. S. Cuadrado, E. Guerra, and J. de Lara. Uncovering errors in ATL model transformations using static analysis and constraint solving. In *ISSRE*, pages 34–44. IEEE, 2014.
3. C. A. Gonzalez, F. Büttner, R. Clariso, and J. Cabot. Emftocsp: A tool for the lightweight verification of emf models. In *Proceedings of the First International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches*, pages 44–50. IEEE Press, 2012.
4. M. Kuhlmann, L. Hamann, and M. Gogolla. Extensive validation of OCL models by integrating SAT solving into USE. In *TOOLS (49)*, volume 6705 of *LNCS*, pages 290–306. Springer, 2011.
5. P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In *Static Analysis*, pages 298–315. Springer, 2011.