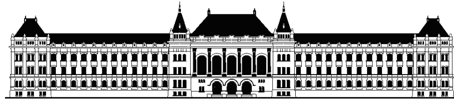

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Verification of Domain Specific Languages

Technical report

Author:
Semeráth Oszkár

Advisor:
dr. Varró Dániel
Associate Professor, BME-MIT

created: October 28, 2012
last edited: March 28, 2013

Contents

1 Preliminaries **2**
1.1 Technological basics . . . 2
1.2 Overview of the approach . . . 3

2 Z3 Representation of the EMF Metamodel **4**
2.1 Objects . . . 4
2.2 Types . . . 4
2.3 Class hierarchy . . . 4
2.4 Reference . . . 5
2.5 Multiplicity . . . 6
2.6 Inverse edges . . . 6
2.7 Containment . . . 7
2.8 Instance models . . . 7
2.9 Conclusion . . . 8

3 Z3 Representation of the IncQuery Patterns **9**
3.1 Structure of the Patterns . . . 9
3.2 Defining Domain Specific Languages with IncQuery Patterns . . . 10
3.3 Classifier Constraints . . . 10
3.4 Path Constraints . . . 10
3.5 Constraint Approximation . . . 11
3.6 Recursive Patterns . . . 11
3.7 Pattern Call . . . 12
3.8 Creating an Approximated Pattern Call Graph . . . 14
3.9 Transitive Closure . . . 14
3.10 Conclusion . . . 17

Bibliography **18**

Chapter 1

Preliminaries

1.1 Technological basics

The design of integrated development environments (IDEs) for complex domain-specific languages (DSL) is still a challenging task nowadays. Generative environments like the Eclipse Modeling Framework (EMF) [10], Xtext or the Graphical Modeling Framework (GMF) significantly improve productivity by automating the production of rich editor features (e.g. syntax highlighting, auto-completion, etc.) to enhance modeling for domain experts. Furthermore, there is efficient tool support for validating well-formedness constraints and design rules over large model instances of the DSL using tools like Eclipse OCL [11] or EMF-IncQuery [3]. As a result, Eclipse-based IDEs are widely used in the industry in various domains including business modeling, avionics or automotive.

However, in case of complex, standardized industrial domains (like ARINC 653 [1] for avionics or AUTOSAR [2] in automotive), the sheer complexity of the DSL is a major challenge itself. (1) First, there are hundreds of well-formedness constraints and design rules defined by those standards, and due to the lack of validation, there is no guarantee for their consistency or unambiguity. (2) Moreover, domain metamodels are frequently extended by derived features, which serve as automatically calculated shortcuts for accessing or navigating models in a more straightforward way. In many practical cases, these features are not defined by the underlying standards but introduced during the construction of the DSL environment for efficiency reasons. Anyhow, the specification of derived features can also be inconsistent, ambiguous or incomplete.

As model-driven tools are frequently used in critical systems design to detect conceptual flaws of the system model early in the development process to decrease verification and validation (V&V) costs, those tools should be validated with the same level of scrutiny as the underlying system tools as part of a software tool qualification process issues in order to provide trust in their output. Therefore software tool qualification raises several challenges for building trusted DSL tools in a specific domain.

In the current paper, we aim to validate DSL tools by proposing an automated mapping from their high-level specification to the state-of-the-art Z3 SMT-solver [4]. We assume that DSL tools are specified by their respective EMF metamodels extended with derived features and well-formedness constraints captured (and implemented) by graph queries within the EMF-IncQuery framework [8, 5]. We define a validation process, which gradually investigates derived features and well-formedness constraints to pinpoint inconsistency, ambiguity or incompleteness issues. We identify constraints and derived features which can be mapped to effectively propositional logic formula [7], which are a decidable fragment of first order logic with effective reasoning support. Moreover, we provide several approximations for constraints which lie outside of this fragment to enable formal analysis of a practically relevant set of constraints.

The main innovation of our approach is to provide a *combined validation* of metamodels, derived features and well-formedness constraints *defined by an advanced graph query language*

(instead of OCL) using *approximations to cover complex query features*. Our approach is illustrated on validating several DSL tool features taken from an ongoing industrial project in the avionics domain.

1.2 Overview of the approach

Our approach (illustrated in Figure 1.1) aims at validating complex DSL languages by automatically mapping from their high-level specification to the Z3 [4] SMT-solver. These complex DSLs are assumed to be defined by (i) a *metamodel* specified in EMF and augmented with both (ii) *derived features* and (iii) *well-formedness* (WF) constraints captured by model queries within the EMF-IncQuery framework. These three artifacts form the input for our generator to provide the logical formulas that is fed into the Z3 solver. The output of the solver is either a *proof of inconsistency* or a *valid model* that satisfies all given constraints generated from the input artifacts.

Additionally, *search parameters* can be defined to impose additional restrictions or specific overapproximations to reduce the complexity of the formula to be proved. Moreover, as an optional input for the generator the user can define – based on the counter examples and proves provided by the solver – specific instance level constraints in the form of an *partial snapshot* [9, 6] (also called input model) to restrict the domain of possible instance model and thus prune trivial valid models (e.g., empty model) provided by the Z3 solver.

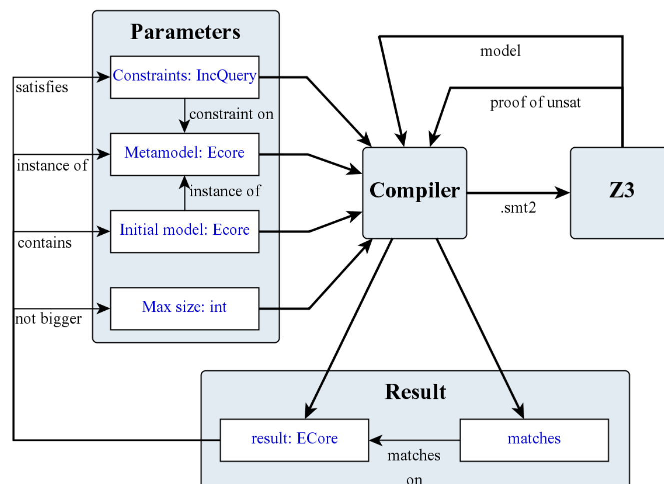

Figure 1.1: *The structure of the analysis*

The tool can be used to:

- check the *consistency* of a domain specific language.
- get a minimal element of a language.
- *formally prove* some property of an informally specified design language.
- compare the matches of the patterns with the IncQuery, and used as a *test oracle*.

Chapter 2

Z3 Representation of the EMF Metamodel

2.1 Objects

The models of the EMF framework are *graph based*, where the EObjects are the nodes of the graphs and the EReferences are the edges. The objects of an EMF model are one-to-one compiled to *individuals* of the Z3 models that instances of a type `Object`.

```
(declare-sort Object)
```

If the number of object is bounded there is a more efficient way to declare fix sized enum-like types:

```
(declare-datatypes () ((Object O1 O2 O3)))
```

where `O1`, `O2` and `O3` are the only objects in an three element model.

2.2 Types

The possible types of the instance model are the classes of the input Ecore metamodel. The language of the Z3 does not support inheritance between the types, so other convention is needed to be used. The most efficient way to describe if an object is an instance of a type from the finite domain of the classifiers is the definition of *type indicator predicates*. If an `o` object is an instance of a type `T`, the `isTypeT o` expression is true, and false otherwise. The declaration of the possible types should be enumerated.

Example 1 For example with types `A`, `B` and `C` it looks like:

```
(declare-fun isTypeA (Object) Bool)
(declare-fun isTypeB (Object) Bool)
(declare-fun isTypeC (Object) Bool)
```

2.3 Class hierarchy

A simple way to define the *type hierarchy* of the EMF is to enumerate the possible type cases of type predicate combination. This is definable by a table where every column representing a type and every row a concrete (nor abstract, neither interface) type. The cells contain a predicate. This predicate is positive if the type of the row is compatible with the type of the column, and negated if it is not. The type `A` is compatible with the type `B` if they satisfy one of those:

- `A`=`B`

- `B` is supertype of `A`

The conjunction of the predicates in a row describes a possible case of the type evaluation of an object. The disjunction of those different type cases expresses that an object satisfies one and only one type case. This feature is expressed to every object.

Example 2 The following example expresses the type hiearchy of the metamodel visible in figure 2.1.

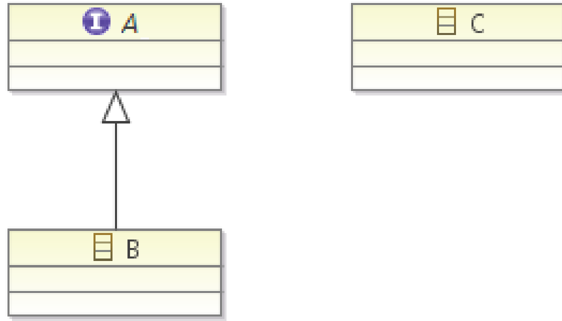

Figure 2.1: *Type hierarchy example.*

And the statement that defines the possible types are the following:

`(assert (forall ((o Object))( or`

		A	B	C	
B	`(and`	`(isTypeA o)`	`(isTypeB o)`	`(not (isTypeC o))`	`)`
C	`(and`	`(not (isTypeA o))`	`(not (isTypeB o))`	`(isTypeC o)`	`)`

`)))`

The size of this statement is $O(|types|^2)$ which seems quiet big, but in case of mostly disjoint types there is not any much more efficient technique: it should be stated that the predicates are mutually exclusive.

If the metamodel extension is enabled the type hierarchy could be extended that the an object could have any type set from the powerset of *types*. The only constraint that the inheritance should be satisfied: if an object has the type `A` and `A` is compatible with the type `B` then the object has the type `B` too. This can be expressed by simple implications over the type predicates.

2.4 Reference

The references of the EMF models are the edges between the object-nodes. Those edges are directed; loop edges are allowed; parallel edges are not supported but not prohibited. The IncQuery pattern language follows the ideology of the EMF. This concludes that the EMF metamodel with IncQuery constraints is *incapable to define* any parallelism property of the instance models. This fact reduces the complexity of references to the level of *relations* which can be easily described as Z3 relations.

The types of the objects on the ends of the relations are needed to be defined. In case of a relation this definition is a simple assumption: if a (`O1`,`O2`) pair satisfies the relation than the `O1` need to be an instance of the source of the relation and `O2` needs to be the instance of the target.

Example 3 The following example defines the types of the references visible on the Figure 2.2:

```
  ; reference A.b
(assert (forall ((s Object) (t Object))(=> (referenceAB s t)
(and (isTypeA s) (isTypeB t)))))
  ; reference B.a
(assert (forall ((s Object) (t Object))(=> (referenceBA s t)
(and (isTypeB s) (isTypeA t)))))
```

2.5 Multiplicity

A relation with `0..*` multiplicity by default. In the other cases, some multiplicity assumption might be necessary.

In case of `n..m` the n lower bound means that every object is in relation with n different target. An m upper bound can be set on an object by stating that there is not $m+1$ different target that is in relation with the object. This can be expressed in E High numbers can only be defined with unacceptable amount of existential quantifier, which can easily kill the efficiency of the theorem prover. This assumption should be added with low priority to the system. Some example:

Example 4 The relations of the figure 2.2 should be declared in the following way (the first is the `A.b` reference, the second is the `B.a`):

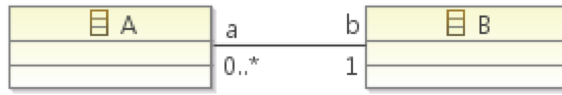

Figure 2.2: *Reference example.*

```
(declare-fun referenceAB (Object Object) Bool)
(declare-fun referenceBA (Object Object) Bool)
```

The multiplicities of `A.b` is implicitly defined. The multiplicities of `B.a` are presented in the following lines (the upper bound have got two equivalent versions: the first is more intuitive and the second is more efficient):

Lower: `(assert (forall ((src Object)) (=> (isTypeA src)`
`(exists ((trg Object)) (referenceAB src trg)))))`

Upper v1: `(assert (forall ((src Object)) (not`
`(exists ((trg1 Object)(trgextra Object)) (and (not (= trg1 trg2))`
`(referenceAB src trg1)(referenceAB src trgextra))))))`

Upper v2: `(assert (forall ((src Object)(trg1 Object)(trgextra Object)) (=>`
`(and (referenceAB src trg1)(referenceAB src trgextra)) (or (= trg1 trgextra)))))`

2.6 Inverse edges

The inverse of a reference can also be defined as demonstrated in the following example:

Example 5 The `A.b` and `B.a` references on figure 2.2 are the inverse of each other. This can be stated by:

```
  (assert (forall ((s Object) (t Object))(=> (referenceAB s t)
(referenceBA t s)))
```

```
  (assert (forall ((s Object) (t Object))(=> (referenceBA s t)
(referenceAB t s)))
```

2.7 Containment

The objects of an EMF model are arranged in a *directed tree hierarchy* by the containment edges. A tree is a connected acyclic graph:

- The acyclicity describable by that any object is unreachable from itself by the path of composition edges. This can be approximanted like as an IncQuery pattern with a simple transitive closure.
- If the previous requirement is satisfied, the composition graph is a DAG. There should be a root element in this graph that referenced by the `root` Z3 constant.
- Every object of the model have exactly one parent in the model with the only exception of the root element.

If these three requirements are held then composition edges are arranged in a tree.

Example 6 At first the the root constant and the containment relation are declared. There is containment relation between two object if and only if there is a containment type edge between them.

```
(declare-fun root () Object)
(declare-fun containment (Object Object) bool)
```

We defined the statement that the roots does not have any parent and each non-root element have exactly one in the following way:

```
(assert (forall ((parent Object)) (not(containment parent root))))
(assert (forall ((o Object)) (or(= o root)( exists ((parent Object))
(and(not (= parent o)) (containment parent o))))))
```

2.8 Instance models

The analysis can be parameterized by an initial instance model of the metamodel. This initial model can be inserted to the axiom system of input of the Z3 with the following translations steps:

- Every object of the instance model is compiled into a constant, like: `(declare-const instance1 Object)`.
- The constants are forced to be unique: `(assert (distinct instance1 instance2 ...))`.
- The type of a constant is settable by the type predicate: `(assert (isTypeA instance1))`.
- The references are settable by the same way: `(assert (referenceAB instance1 instance2))`.

2.9 Conclusion

The generated statements defines the basic structure of the analysed domain. The set of those statements are handled as *META*.

In general, this report aims to ensure the *traceability* between the two kinds of model. This means that every distinct valid Z3 model can be translated to every different EMF instance model. This requirement is satisfied at metamodel mapping.

The second goal is to achieve decidability and efficient feasibility if it is possible with the given domain. For this purpose a subclass of the first order logic is used:

Definition 1 (Effectively propositional logic) *The effectively propositional logic[7] is a fragment of the first order logic, which contains finite number of constant variables and the statements in prenex form build from universal quantifiers, predicates and logical connectives.*

This fragment of the first order logic is *decidable*, and efficiently provable.

The list of features of an EMF metamodel and witch one is expressible at effectively propositional form:

Section	Name		
section 2.1	Objects	E	+
section 2.2	Types	E	+
section 2.3	Class hierarchy	E	+
section 2.4	References	E	+
section 2.5	`0..X` multiplicities	E	+
section 2.5	Not `0..X` multiplicities (like `1..1`)	E	−
section 2.6	Inverse edges	E	+
section 2.7	Containment hierarchy	A	−
section 2.8	Initial instance model	E	+

where:

E	means that the feature is expressible
A	means that the feature can be approximated (see the exact definition later)
+	means that the expression is at effectively propositional form
−	means that the expression is not at effectively propositional form

Chapter 3

Z3 Representation of the IncQuery Patterns

3.1 Structure of the Patterns

An IncQuery pattern consists of a *parameter list* and some *pattern body*. The parameter list is a fix sized vector of variables over the EObject-s of the model, let sign this with $Params$. The bodies define *properties* of the parameters; this definition may use some other existentially quantified object *variables* called $Vars$. The properties are defined in by *constraints* over the parameters and the variables. The disjunction of the properties of the bodies define the predicate of the pattern. Formally:

$$pattern(Params) = \bigvee_{\substack{body \in \\ pattern.\texttt{bodies}}} \exists Vars \bigwedge_{\substack{constraint \in \\ body.\texttt{constraints}}} constraint(Params, Vars)$$

The *match-set* of a pattern is a relation over the objects of the model. An object vector is in the match-set if and only if the vector satisfies predicate of the pattern:

$$Params \in patternMatch \Leftrightarrow pattern(Params)$$

The relation of a pattern is defined as a predicate over the objects of the model. The interpretation of the pattern matching predicates are enforced by *implication assertions*. Some cases a direction of implication might be omitted if it does not serve any cause in the analysis.

Example 7 There is a two parametered pattern called `Reachable(x, y)`. The matchings of this pattern are defined by the satisfaction of the following predicate:

```
(declare-fun patternReachable (Object Object) Bool)
```

If some property defined in the bodies of the pattern holds, the predicate should be true, and the satisfaction of the predicate implies the mentioned property:

$$Params \in patternMatch \Leftrightarrow pattern(Params)$$

```
(assert (forall ((x Object)(y Object))( iff (properties)
(patternReachable x y))))
```

3.2 Defining Domain Specific Languages with IncQuery Patterns

An M model is the element of a *domain specific language* (DSL) if it is conform with the metamodel ($META$) and satisfies its well-formedness statements (WF):

$$M \in DSL \Leftrightarrow META \wedge WF \models M$$

The goal of the validation framework of the IncQuery is to define the well-formedness by patterns. The model is *violating* the well-formedness (WF) criteria if it is satisfying any of the *forbidden pattern* (that is marked with `@Constraint` annotation). So a well formed M is free from any matches of those patterns:

$$M \models WF = \neg \bigvee_{\substack{pattern \in \\ \text{annoted patterns}}} \exists Params\ pattern(Params)$$

Develop the patterns and the vectors of the WF statement:

$$WF = \neg \bigvee_{patterns} \underbrace{\exists p_1 \ldots \exists p_n}_{\text{parameters}} \bigvee_{bodies} \underbrace{\exists v_1 \ldots \exists v_n}_{\text{variables}} \bigwedge_{constraints} constraint(Params, Vars)$$

Which equals with this in prenex form (renaming the parameters and the variables to unique name might be necessary) :

$$WF = \underbrace{\forall p_1 \ldots \forall p_n}_{\text{parameters}} \underbrace{\forall v_1 \ldots \forall v_n}_{\text{variables}} \bigwedge_{patterns} \bigwedge_{bodies} \bigvee_{constraints} \neg constraint(Params, Vars)$$

And that form is satisfying in the requirements of the the of the effectively propositional logic.

Statement 1 (Effectively propositional nature of patterns) *If every constraint in the patterns are definable in a form of predicate that does not use any universally quantified variable then the well-formedness is an effectively propositional statement.*

The aim is to keep the effective propositional property of the Z3 statements whenever it is possible. The following sections provide the translation of the constrains of the IncQuery language to a Z3 expression.

3.3 Classifier Constraints

A *classifies constraint* defines the type of the objects that are binded to the variable. The IncQuery constraint can be easily compiled to type predicate.

Example 8 There is an IncQuery classifier constraint that defines that the variable x is an instance of the type T: `T(x)`. In Z3 the constraint is compiled to this expression: `isTypeT x`.

3.4 Path Constraints

Path constrains in IncQuery defines that there is a path consists of sequence of references from the defined type that leads from a variable to another. By introducing the implicit object variables as the inner nodes of the path, the expression can be compiled into simple reference requirements.

Example 9 The IncQuery path expression constraint `A.b.c.d(x,y)` defines that there is a path that starts from x, leads through references b, c and d and ends in the y object. The path

touches some further object that should be referred by implicit variables. In this example there is two of that that should be called `pathVariable1` and `pathVariable2`.

After this the following sequence defines the existence of the path:

```
and (referenceAB x pathVariable1)
(referenceBC pathVariable1 pathVariable2) (referenceCD pathVariable2 y)
```

The equivalence and unequivalence of two object can be simply defined as well.

The check expressions are relations over the variables that are impossible to be fully supported. If the results of the case expressions are modelled by a predicate than the result could be consequent. In individual cases some manual assertion might be achievable.

3.5 Constraint Approximation

The expressive power of the IncQuery language is definitely larger than the input language of the Z3 first order theorem prover. Some constraints such as recursively called patterns, transitive closures, set cardinalities and check expressions can not be fully compiled into it.

To handle this some approximation techniques need to be deployed:

Definition 2 (Approximations of Constraints) *The C_{UA} predicate is underapproximate (C_{OA} overapproximate) the C constraint if it satisfies the following implications for every parametrisation:*

$$C_{UA} \Rightarrow C \quad (C \Rightarrow C_{OA})$$

The constant $true$ predicate is always a good overapproximation, and $false$ approximates every predicate under. A statement also approximates itself. Let us see what approximated constraints can provide:

Definition 3 (Approximations of Well-Formedness) *The axioms of well-formedness WF of a domain specific language are overapproxiated (underapproxiamted) by WF_{OA} (WF_{UA}) if it is produced by the overapproximation (underapproxiamtion) of the forbidden patterns defining the WF.*

The approximations define languages with more or less elements than the approximated one, as the following implications show:

$$META_{UA} \wedge WF_{UA} \models M \Rightarrow META \wedge WF \models M$$

$$META_{OA} \wedge WF_{OA} \not\models M \Rightarrow META \wedge WF \not\models M$$

If the analysis is about the consistency of the language, the following implications are the interesting ones:

$$META_{OA} \wedge WF_{OA} \text{ unsatisfiable} \Rightarrow META \wedge WF \text{ unsatisfiable}$$

$$META_{UA} \wedge WF_{UA} \text{ satisfiable} \Rightarrow META \wedge WF \text{ satisfiable}$$

This means that the consistency check of a domain specific language can be done by verifying a more general logical structure what more efficient to reason over.

3.6 Recursive Patterns

The pattern call constraint makes it possible to compose more complex patterns that referring to others. Those complex patterns might exceed the expression power of the Z3, so it needed to be handled carefully.

The Z3 theorem prover does not support the definition of recursive axioms, but the IncQuery language does. However, some experimental test has been tried about the recursive axioms that resulted in the following results:

- sometimes the axioms are handled correctly.
- if the Z3 has reached some kind of fix point in the proving, the created model are believed to be correct wrongly.
- ff the axiom system is inconstant the model search resulted in infinite loop.

This means that results are overapproximations of the domain specific language, but it fail to work when this kind of approximations are useful. The developers promised that this feature will be supported, but not in the near future. This implied that the recursion must be handled manually.

3.7 Pattern Call

The conclusion of the previous section is that the recursive pattern calls have to be approximated. At first the cases of recursion are needed to be found. For this, a pattern call graph should be created witch sign every pattern call constraints.

Definition 4 (Call Graph) *A pattern call graph is a directed graph, where every node is bijectively mapped to a pattern (representing* $: V \mapsto Patterns$*), and there is a directed edge between two nodes if and only if the pattern of the source edge contains a call to the pattern of the target of the edge.*

Every circle in the call graph can be traced back to a case of a recursion. So the goal is to create an acyclic call graph that somehow approximate the original call graph.

The approximated call graph is need to be introduced. The first difference between this and the original is that a pattern might be represented by multiple node in the call graph, but one node must be assigned as an entry point. The second is that there might be omitted cases of the pattern calls. Formally:

Definition 5 (Approximated Call Graph) *Approximated call graph is a directed acyclic graph (DAG), where every node is surjectively mapped to a pattern (representing* $: V \mapsto Patterns$*), every pattern are injectively mapped to a node (*$entry : Patterns \mapsto V$*), and if there is an edge between two nodes that means that the pattern of the source edge contains a call to the pattern of the target of the edge.*

Example 10 Check the following IncQuery patterns:

```
pattern A(...) {find B(...); ...}
pattern B(...) {find C(...); find D(...);...}
pattern C(...) {find A(...); ...}
pattern D(...) {...}
```

The call graph of this is visible on the left side of the figure 3.1.

There is an approximation of this call graph on the right side of this figure. The nodes labelled with a pair of a letter and a number representing the patterns with the same name as the letter. There are different nodes that representing the same pattern they are distinguished by the numbers. The ones with the number 0 are the entry points. Note that the edges representing the calls can leads into any instance of the node belongs to the target pattern as long as the acyclicity is hold. There is a missing call from `C2` to an `A` node that is allowed. (It needs to be allowed or else the DAG property can not be satisfied.)

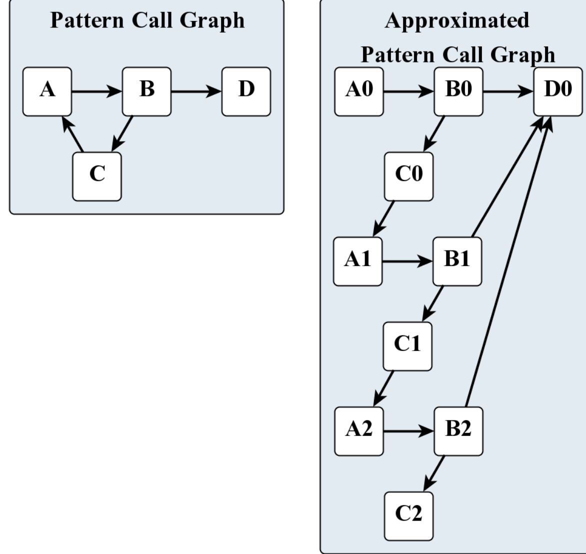

Figure 3.1: *Normal and approximated call graphs*

When an approximated call graph is constructed than the approximations of the pattern call constraints can be created. For each node of the approximated call graph a new pattern matching relation should be created, like it was a whole new pattern. In the property assertions of those patterns, every pattern call constraint should refer to the relation that belongs to the target of the pattern call edge. If there is a pattern call constraint that has not got edge in the approximated call graph, the constraint should be approximated to a constant *true* or *false* predicate.

A pattern body is the conjunction of the constraints. This means that a constant *true* predicate is omittable without changing the semantics, and a constant *false* predicate make the body unsatisfiable,

This means that the unapproximated pattern match relation has to satisfy all the over- and underapproximation property. The following assertion was the inexpressible one, where *pattern* was possibly recursive:

$$Params \in patternMatch \Leftrightarrow pattern(Params)$$

The $pattern(Params)$ property has been approximated:

$$pattern_{UA}(Params) \Rightarrow pattern(Params) \Rightarrow pattern_{OA}(Params)$$

So the following assertion defines the relation of the match-set are under- and overapproximated by the properties defined by the pattern:

$$pattern_{UA}(Params) \Rightarrow Params \in patternMatch \Rightarrow pattern_{OA}(Params)$$

Example 11 We want to define the pattern referred in the pattern call graph in figure 3.1, and we only need overapproximations. For this purpose we use the approximated pattern call graph on the right side of same figure.

First we define the pattern matching relations:

(declare-fun A (...) Bool) (declare-fun A0 (...) Bool) (declare-fun A1 (...) Bool) (declare-fun A2 (...) Bool)	(declare-fun B (...) Bool) (declare-fun B0 (...) Bool) (declare-fun B1 (...) Bool) (declare-fun B2 (...) Bool)
(declare-fun C (...) Bool) (declare-fun C0 (...) Bool) (declare-fun C1 (...) Bool) (declare-fun C2 (...) Bool)	(declare-fun D (...) Bool)

After this, the approximated pattern properties can be added. The target og the pattern calls are defined in the approximated pattern call graph:

`A`: approximated in `A0`: referring to `B0` in `A1`: referring to `B1` in `A2`: referring to `B2`	`B`: approximated in `B0`: referring to `C0` and `D0` in `B1`: referring to `C1` and `D0` in `B2`: referring to `C2` and `D0`
`C`: approximated in `C0`: referring to `A1` in `C1`: referring to `A2` in `C2`: *true*	

And the approximations:

A $\Rightarrow$ A0	B $\Rightarrow$ B0
C $\Rightarrow$ C0	D $\Leftrightarrow$ D0

3.8 Creating an Approximated Pattern Call Graph

In this section I provide an algorithm that can create an approximated pattern call graph.

Let $MaxCallNumber \in \mathcal{N}$ mark Level of precision in the approximation. The base idea of this approximation is that it neglects the cycles in the call graph where a pattern has been called more times than the $MaxCallNumber$ parameter.

The algorithm is a repetition bounded depth-first search. It starts on a pseudo-node that calls the forbidden patterns. A call history is a multiset of patterns; in this collection shows that a pattern is how many times called before an actual call. The algorithm: 1.

3.9 Transitive Closure

The Z3 theorem prover is a highly optimised tool. If the approximations can be failed easily, the prover will try to do it, because the failed approximations do not enforce anything. This happens for example when transitive closure simply written in form of recursion is tried to be overapproximated.

Example 12 For the sake of simplicity the following example is demonstrated in the language of the IncQuery patterns. The goal is to define reachability by recursion. (Usually transitive closure is recommended for this purpose.) It would look like this:

```
pattern reachable(x, y) {
Node.edge(x,y); } or {
Node.edge(x,middle);
```

Algorithm 1: CreateApproximationNode

Data: n: Node, h: Call history
Result: a: Approximated node
a := new Approximation node
$a.hasEnded := false$
$a.representing$:= pattern of n
begin
  if *h does not contains $a.representing$* **then**
    $entry(a.representing)$:= a
  end
  while *n has unprocessed neighbour m* **do**
    if *$entry(m).hasEnded$* **then**
      `If the control is here than` h `does not contains` $entry(m)$.
      add an edge between a and $entry(m)$
    else
      $h2$:= h + $a.representing$
      if *every element in $h2$ has multiplicity up to MaxCallNumber* **then**
        add an edge between a and $CreateApproximationNode(m, h2)$
      end
    end
  end
  $a.hasEnded := true$
end

```
find reachable(x,y);
}
```

When overapproximated for example at most 3 steps, the following patterns would be compiled:

```
pattern reachableOrCanStepAtLeast3(x, y) {
      Node.edge(x,y); } or {
      Node.edge(x,middle);
      find reachableOrCanStepAtLeast2(middle,y);
}
pattern reachableOrCanStepAtLeast2(middle, y) {
      Node.edge(x,y); } or {
      Node.edge(x,middle);
      find reachableOrCanStepAtLeast1(x,y);
}
pattern reachableOrCanStepAtLeast1(middle, y) {
      Node.edge(x,y); } or {
      Node.edge(x,middle);
      find reachableOrCanStepAtLeast0(middle,y);
}
pattern reachableOrCanStepAtLeast0(x, y) {
      Node.edge(x,y); } or {
      Node.edge(x,middle);
      // find reachable(middle,y) approximated to true
}
```

Get an example of at most 100 steps. The expected behaviour of the prover is that it tries to

find reachable nodes, and if it stuck in 100 steps it invalidate the unapproximated property.

In reality that happens what the figure 3.2 shows when the Z3 asked to overapproximate the reachability between `A` and `E`: the tool finds a circle and wastes all the approximation steps in a 3 long cicle.

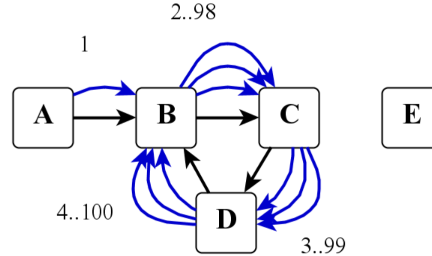

Figure 3.2: *The behaviour of naively overapproximated transitive closure*

The first moral of the example is to forcing the tool to effectively approximate a recursive pattern is might be hard, and it is not likely be automatizabled. The second is that the approximation of the transitive closure is needed to be with simple paths rather than paths.

The difference between simple paths and paths that the simple ones has unique nodes. This should be enforced in the patterns by inequality constraints: the variable named `middle` equivalent to:

- the source or the target of the transitive closure
- earlier elements of the path

The earlier elements can be filtered if the pattern parametrized by those elements.

Example 13 A valid implementation of the approximation of the transitive closure: `pattern`

```
reachableOrCanStepAtLeast3(x, y) {
	Node.edge(x,y); } or {
	middle != x;
	middle != y;
	Node.edge(x,middle);
	find reachableOrCanStepAtLeast2(middle,y,x);
}
pattern reachableOrCanStepAtLeast2(x, y, earlier1) {
	Node.edge(x,y); } or {
	middle != x;
	middle != y;
	middle != earlier1;
	Node.edge(x,middle);
	find reachableOrCanStepAtLeast1(middle,y,earlier1,x);
}
pattern reachableOrCanStepAtLeast1(x, y, earlier1, earlier2) {
	Node.edge(x,y); } or {
	middle != x;
	middle != y;
	middle != earlier1;
	middle != earlier2;
```

```
        Node.edge(x,middle);
        find reachableOrCanStepAtLeast0(middle,y,earlier1,earlier2,x);
}
pattern reachableOrCanStepAtLeast0(x, y, earlier1, earlier2, earlier3) {
        Node.edge(x,y); } or {
        middle != x;
        middle != y;
        middle != earlier1;
        middle != earlier2;
        middle != earlier3;
        Node.edge(x,middle);
        // find reachable(middle,y) approximated to true
}
```

3.10 Conclusion

The axiom system of the input consists of the statements derived from the metamodel, the initial instance model and the forbidden patterns. The following features are expressible in effectively propositional form:

section 3.3	Classifier constraint	E	+
section 3.3	Classifier constraint from parameter list	E	+
section 3.4	Path constraint	E	+
section 3.7	Positive non-circular call	E	+
section 3.7	Negative pattern call	E	−
section 3.6	Positive recursion	A	+
section 3.9	Transitive closure	A	+
section 3.7	Arbitrary call of patterns	A	−
	Cardinality	X	
	Check expressions	X	

where:
- E means that the feature is expressible
- A means that the feature can be approximated
- X means that feature is inexpressible
- + means that the expression is at effectively propositional form
- − means that the expression is not at effectively propositional form

The traceability is also provided with the exception of the recursive patterns: in this case the patterns may have multiple valid case of different match set. The IncQuery chooses the the match-set depending on the history of the edited model. The compiler can be used to detect such cases, where the output of the pattern matcher is ambiguous.

Bibliography

[1] ARINC - Aeronautical Radio, Incorporated. A653 - Avionics Application Software Standard Interface.

[2] AUTOSAR Consortium. *The AUTOSAR Standard.* http://www.autosar.org/.

[3] Gábor Bergmann, Ákos Horváth, István Ráth, Dániel Varró, András Balogh, Zoltán Balogh, and András Ökrös. Incremental Evaluation of Model Queries over EMF Models. In *MODELS'10*, volume 6395 of *LNCS*. Springer, 2010.

[4] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340. Springer-Verlag, 2008.

[5] Ábel Hegedüs, Ákos Horváth, István Ráth, and Dániel Varró. Query-driven soft interconnection of EMF models. In *Proc of the Int. Conf on Model Driven Engineering Languages and Systems*, volume LNCS 7590, pages 134–150, 2012.

[6] Ethan K. Jackson, Tihamer Levendovszky, and Daniel Balasubramanian. Reasoning about metamodeling with formal specifications and automatic proofs. In *Proc. of the 14th Int. Conf. on Model Driven Engineering Languages and Systems*, volume 6981 of *LNCS*, pages 653–667, 2011.

[7] Ruzica Piskac, Leonardo de Moura, and Nikolaj Bjorner. Deciding effectively propositional logic with equality, 2008. Microsoft Research, MSR-TR-2008-181 Technical Report.

[8] István Ráth, Ábel Hegedüs, and Dániel Varró. Derived features for EMF by integrating advanced model queries. In Antonio Vallecillo, Juha-Pekka Tolvanen, Ekkart Kindler, Harald Störrle, and Dimitris Kolovos, editors, *Modelling Foundations and Applications*, volume 7349 of *Lecture Notes in Computer Science*, pages 102–117. Springer Berlin / Heidelberg, 2012.

[9] Sagar Sen, Jean-Marie Mottu, Massimo Tisi, and Jordi Cabot. Using models of partial knowledge to test model transformations. In *5th Int. Conf. on Theory and Practice of Model Transformations*, volume 7307 of *LNCS*, pages 24–39, 2012.

[10] The Eclipse Project. *Eclipse Modeling Framework.* http://www.eclipse.org/emf.

[11] E. D. Willink. An extensible OCL virtual machine and code generator. In *Proc. of the 12th Workshop on OCL and Textual Modelling*, pages 13–18. ACM, 2012.