Grammar-based Program Generation Based on Model Finding

Mathias Soeken Rolf Drechsler

Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{msoeken,drechsle}@informatik.uni-bremen.de

***Abstract*—This paper presents an algorithm that generates test programs in order to test programming languages and domain specific languages using formal methods. The novelty of the approach is that it is embedded into a model driven engineering environment and it is described as a model finding problem. The grammar of the language and the respective test programs are represented as meta-models and models, respectively. As a result, model finders are utilized to generate test programs based on user constraints while additionally ensuring embedded constraints of the programming languages. An experimental evaluation demonstrates the applicability of the approach.**

I. Introduction

Recently model finders have become significantly attractive in the field of *Model Driven Engineering* (MDE). Leveraging the achievements in automatic proof techniques, model finders became applicable for solving verification tasks on meta-models (e.g. [1]–[3]). Model finders require a meta-model and a list of constraints and return a model that conforms to the meta-model and meets the given constraints, if such a model exists. Otherwise, the absence of a model adhering to the constraints is proven.

Furthermore, model driven engineering techniques have found application in many aspects of software engineering. One particular example is Xtext, a parser generator that integrates well into the *Eclipse Modeling Framework* (EMF), an implementation of the *Meta Object Facility* (MOF) [4] based on the Eclipse IDE. In Xtext, the grammar description is transformed into a meta-model and each program corresponds to a model that conforms to that meta-model. The model can also be seen as the in-memory abstract syntax tree.

However, due to the explicit use of meta-models and models, model finders can be directly applied. Since models correspond to programs of the respective grammar, the result of the model finder is a program. In other words, in the context of Xtext and grammars as meta-models, model finders become "program finders."

Many approaches for grammar-based program generation have been proposed in the past, e.g. [5]–[10]. The aim of this paper is not to compete with these algorithms. Instead, we show that the problem can be described in terms of a model finding application therefore demonstrating the generality of these methods.

The applicability of the proposed approach has been evaluated with experiments based on the languages CoffeeScript, Lua, and BibTeX. The results of the evaluation show that many test programs can be generated efficiently.

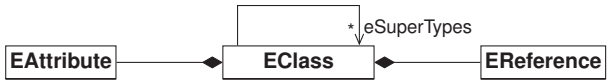

Fig. 1. Subset of the Ecore meta-meta-model

The paper is structured as follows. In the following section, preliminaries provide the necessary background that is required for the paper. Section III illustrates the general idea, while precise application scenarios are given in Section IV. In Section V, the results of the experimental evaluation are presented before the paper is concluded in VI.

II. Preliminaries

In order to keep the paper self-contained, this section provides the necessary background by outlining the Eclipse Modeling Framework, Xtext, the Object Constraint Language, and model finding.

A. Eclipse Modeling Framework

The Eclipse Modeling Framework [11] provides modeling languages and tools in order to facilitate the development of applications and work flows based on structured model data. As an example, the EMF provides textual and graphical editors to design meta-models and their models, as well as algorithms for code generation or generation of basic editors. Initially aiming at providing an implementation of the Meta Object Facility [4], the EMF contains its own meta-meta-model called Ecore which can be seen as an implementation of the *Essential MOF* (EMOF) [4].

A subset of the Ecore meta-meta-model that is relevant for ongoing discussions is depicted in Fig. 1. The central element is a class (*EClass*) that describes an atomic entity and holds data by means of attributes (*EAttribute*) and references (*EReference*) to other classes. Polymorphism is modeled with the *eSuperTypes* reference. Since an arbitrary number of super types can be specified, multiple inheritance can explicitly be modeled with the Ecore meta-meta-model. Notice that the Ecore meta-meta-model is modeled by its own description means, i.e. all three classes in Fig. 1 are in fact instances of an *EClass*.

978-1-4799-3525-3/13/$31.00 ©2013 IEEE

B. Xtext

The Xtext Eclipse plugin [12] is helpful when implementing programming languages or domain specific languages (DSLs) by covering all aspects of a complete language infrastructure. The plugin can generate a parser, a linker, compilers, and interpreters, as well as IDE elements such as editors, syntax highlighting, validation, and auto checks.

Xtext's grammar syntax is based on ANTLR [13] and in fact ANTLR is used for a variety of tasks under the hood. An example of such a grammar for a simple DSL is given in Fig. 2. This DSL allows for writing statements (Line 5) to declare variables (`VarDecl`, Line 8), to print variables (`PrintVar`, Line 10), and to store variables in a file (`SaveVar`, Line 12). Notice that no semantics has been defined in the grammar. Choices are denoted by '`|`' and non-terminals by quoted strings. Furthermore, `ID`, `INT`, and `STRING` are predefined rules that match identifiers, integers, and strings, respectively. Within rules, variable assignments provide identifiers to access the individual elements of a rule, e.g. `filename` in the rule `SaveVar`.

Xtext can be well integrated into MDE work flows, since besides all the language artifacts mentioned above, also a meta-model for the grammar is automatically inferred. The abstract syntax tree when parsing a source file is an in-memory model that conforms to this meta-model. For this purpose, Xtext makes use of the EMF. Roughly speaking, each rule is translated to an *EClass*, each property to an *EAttribute*, and each reference to another rule as an *EReference* based on the Ecore language.

The Ecore meta-model for the grammar defined in Fig. 2 is visualized in Fig. 3. As can be seen, each rule is represented as a class and a reference has been inferred from the relation between the program and statements. Further, the inheritance relation between `Statement` and `VarDecl`, `PrintVar`, and `SaveVar` has been detected. Also the name property which all three statements have in common is automatically detected and therefore added as an attribute to the common parent class. In a similar fashion, attributes are generated for the value in the `VarDecl` rule as well as for the file name in the `SaveVar` rule.

```
1  grammar org.example.MiniDSL with
2          org.eclipse.xtext.common.Terminals
3  generate miniDSL "http://example.org/MiniDSL"

4  Program:
5      statements += Statement+;

6  Statement:
7      VarDecl | PrintVar | SaveVar;

8  VarDecl:
9      name = ID "=" value = INT;

10 PrintVar:
11     "print" name = ID;

12 SaveVar:
13     "save" name = ID "to" filename = STRING;
```

Fig. 2. Xtext grammar for a simple domain specific language

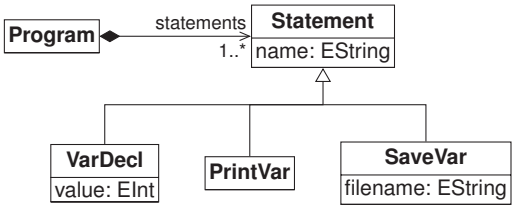

Fig. 3. Meta-model generated from grammar in Fig. 2

In summary, the model hierarchy imposed by the Xtext framework is outlined in Fig. 4. Each grammar is an instance of the Ecore model and each program is an instance to its respective grammar.

C. Object Constraint Language

Meta-models such as the ones built with the EMF and introduced in the previous sections can additionally be extended by textual constraint given in the *Object Constraint Language* (OCL) in order to restrict the set of valid models that can be instantiated from it [14]. OCL offers a variety of functions and notations in order to write constraints and object query expressions on model elements that cannot be described by means of the modeling language, which is often only available in a diagrammatic representation. Plugins such as *OCLinEcore* [15] allow for a convenient integration within other modeling tools.

As an example, the constraint

```
context Program inv decl:
  statements->select(not oclIsTypeOf(VarDecl))
            ->forAll(s | statements->exists(s2|
                s2.oclIsTypeOf(VarDecl) and
                s.name = s2.name))
```
(1)

expresses that all statements which are not variable declarations must refer to declared variables. In detail, first a subset of all statements consisting of only `PrintVar` and `SaveVar` statements is obtained using the *select* function. For all elements in this subset, there must exists another statement which is of type `VarDecl` such that the variable names match.

We distinguish between *integrity constraints* which are constraints that are part of the meta-model and must hold for any instantiated model and *additional constraints* that must be only valid under certain local considerations. The constraint in Equation (1) is an integrity constraint, since for each model it

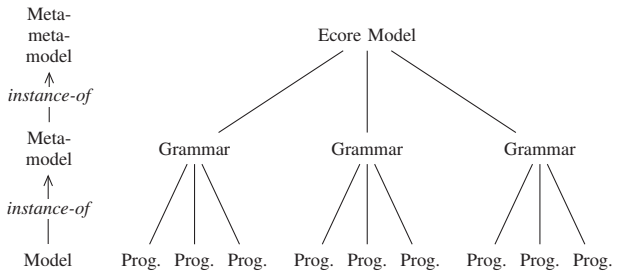

Fig. 4. Model hierarchy

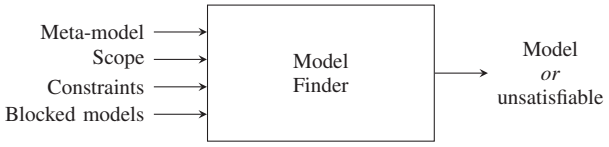

Fig. 5. Model finder

must hold that every used variable is being defined. In contrast to that, the constraint

```
context VarDecl inv positive:
  value > 0
```
(2)

is an additional constraint that requires all variables being initialized with positive values.

D. Model Finding

Given a meta-model and additional constraints, a *model finder* returns one model that meets the additional constraints or proves that no such model exists. In order to execute a model finder automatically also a scope has to be given as input, i.e. bounds for possible object instances and values for data types. This is necessary as otherwise the problem of finding a model becomes undecidable [16]. As an example, one could restrict the scope to 100 statements and 32-bit integers when performing model finding on the meta-model in Fig. 3.

Model finders have been implemented using different strategies such as Boolean satisfiability [1], [17], SMT [2], or constraint satisfaction problems [18]. Although mainly focusing on UML [19], the approaches can easily be applied to similar languages such as EMF, as e.g. demonstrated in [20].

In the scope of this work we consider model finders as a black box as depicted in Fig. 5. Besides the meta-model, constraints, and scopes, a further input is given by a set of blocking models. Since a model finder can only find one model, found models are added to the set of blocking models in order to iteratively find several solutions. The user has no influence on the found solution other than by constraints and blocking models. This solution can be arbitrary, in particular since the model finder depends on the underlying proof algorithms which in turn often make use of random numbers when traversing the search space.

III. Overall Flow

Fig. 6 outlines the overall flow for the proposed algorithm using paper sheets and gears to depict files and programs, respectively. The large box encloses the algorithm that takes two inputs and produces one output. More precisely, given

- a *grammar* that describes the syntax of the programming language
- and additional *constraints* that describe properties,

test programs are automatically generated which meet the given constraints. For this purpose, first a meta-model is automatically deduced from the grammar description of the programming language as it has been demonstrated in Section II-B. This meta-model is given as input to a model finder

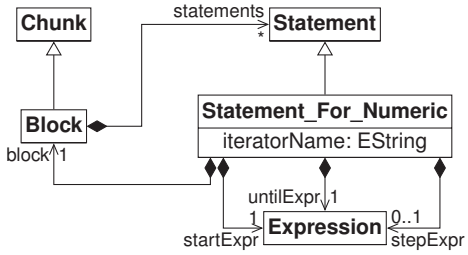

Fig. 7. Excerpt of the Lua grammar as meta-model

together with the additional OCL constraints which describe the constraints for the desired test case. If the model finder cannot find a satisfiable solution it can be concluded that no program exists that fulfills the constraints. In this case, either the constraints were formulated too strong or there is a bug in the grammar representation. If however, the model finder returns a valid model, it can be transformed into a program of the tested programming language using Xtext.

Please notice that the OCL constraints are not only constraints on the deduced meta-model, but also on the grammar itself, in particular since all names are preserved when mapping rules and variables to classes and attributes or references, respectively. As a result, the developer does not need to know any properties of the meta-model which stays hidden in the algorithm and is only used internally.

IV. Program Constraints

This section illustrates how OCL constraint can be used in order to generate test programs by making use of the grammar and the model finder. We do this exemplary using two use cases, nested for loops and statement diversity.

A. Nested For Loops

The first use case describes how to obtain nested for loops using OCL constraints. For this purpose we use the Lua grammar of which the relevant excerpt is given in Fig. 7. A program is organized in chunks, where chunks and blocks have in principle the same syntax, just share different names. A chunk can be seen as the entry point of a program which is nothing else but an executable block. Each block has a possible non-empty list of statements and one particular statement is the numeric for loop. It consists of an iterator name and three expressions for initializing the iterator (startExpr), querying the iterator (untilExpr), and updating the iterator (stepExpr). The syntax of a numeric for loop is given as:

```
for iteratorName = startExpr,untilExpr,stepExpr do
  block
end
```

Given a scope of one chunk, n blocks (including the chunk), and $n-1$ numeric for statements, applying the model finder to the Lua grammar with the constraint

```
context Chunk inv nested:
 Block.allInstances()->one(statements->size()=0)
```
(3)

yields test programs which have $n-1$ nested numeric for loops.

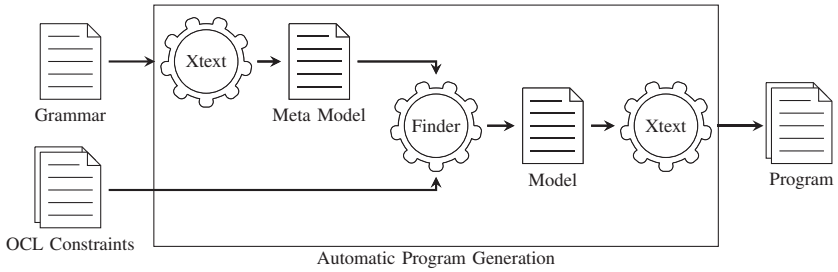

Fig. 6. Overall flow of the proposed algorithm

Notice that the constraint ensures the nested for loop structure, however, all other elements of the test program such as the iterator name or loop expressions can be arbitrary assigned by the model finder. Calling the model finder in an iterative manner while additionally blocking previously found solutions automatically yields different test programs that share the property of having nested for loops.

B. Statement Diversity

Next, programs should be generated that consist of a diversity of statements. More precisely, each block in a program should consist of a defined minimum of statements which should all be of different kinds. It turns out that this task is not as easy as the previous one and requires some modification of the grammar meta-model. However, the modification of the meta-model is an easy step in Xtext as it has special post-process methods that can be integrated into the work flow.

We are adding an integer attribute type to the class *Statement* that is the common super class for each statement. Then, each statement specialization is enhanced with an integrity constraint as follows:

```
context Statement_For_Numeric inv t: type = 0
context Statement_For_Generic inv t: type = 1
context Statement_While       inv t: type = 2
context Statement_Repeat      inv t: type = 3
...
```

This ensures that each statement has a different type, which is necessary as it is not possible to compare the types in OCL expressions directly. Given the extension of the meta-model by the type attribute and the respective integrity constraints programs with diverse statements can be generated e.g. by using the following additional constraint:

```
context Block inv diverse:
  statements->size() >= 3 and
  statements->forAll(s1, s2 |
    s1 <> s2 implies s1.type <> s2.type)
```

It states that each block must have at least three statements and that the types of these statements must be unique.

As can be seen, constraints can be posed using only a few short OCL expressions. Sometimes it might be necessary to extend the meta-model which is a simple task when using Xtext's extension methods.

TABLE I
EVALUATED PROGRAMMING LANGUAGES

Language	Rules	References	Variables
CoffeeScript	66	76	14
Lua	59	75	15
BibTEX	40	73	26

TABLE II
EXPERIMENTAL RESULTS

Language	Benchmark	Run-time (secs)
CoffeeScript	Nested For ($n = 25$)	0.46
CoffeeScript	Nested For ($n = 26$)	0.48
CoffeeScript	Nested For ($n = 27$)	0.53
CoffeeScript	Nested For ($n = 28$)	0.57
CoffeeScript	Nested For ($n = 29$)	0.61
CoffeeScript	Nested For ($n = 30$)	0.63
Lua	Nested For ($n = 25$)	0.66
Lua	Nested For ($n = 26$)	0.68
Lua	Nested For ($n = 27$)	0.71
Lua	Nested For ($n = 28$)	0.74
Lua	Nested For ($n = 29$)	0.80
Lua	Nested For ($n = 30$)	0.81
BibTEX	Diversity ($n = 16$)	0.36
BibTEX	Diversity ($n = 17$)	0.41
BibTEX	Diversity ($n = 18$)	0.43
BibTEX	Diversity ($n = 19$)	0.47
BibTEX	Diversity ($n = 20$)	0.50
BibTEX	Diversity ($n = 21$)	0.54
BibTEX	Diversity ($n = 22$)	0.62
BibTEX	Diversity ($n = 23$)	N/A

V. EXPERIMENTAL EVALUATION

We have implemented the proposed approach using Xtext 2.3.1 [12] and ocl2smt [2] as model finder using Z3 [21] as back-end solver. As languages CoffeeScript [22], Lua [23], and BibTEX [24] were used. Their grammar characteristics such as number of rules, number of references to other rules, and local variables (such as integers and strings) are listed in Table I.

For CoffeeScript and Lua we have conducted the "nested for" experiment that has been described in Section IV-A whereas for BibTEX we have generated instances which cover many different entry types as described in Section IV-B.

All results are listed in Table II. The configuration parameter n describes the number of nested loops and the number of overall entries for the nested for and diversity experiments, respectively. As can be seen, each experiment was processed within less than a second, hence the approach can generate test

programs in a reasonable run-time. The model finder based approach should not be expected to scale better than random test program generators. We have run into problems when trying to generate large instances for the BibTEX diversity benchmarks as the model finder could not create the instance. However, this is caused by limitations in the implementation of the model finder and should be resolved in the future.

VI. Conclusions

We have proposed an approach that facilitates model finding algorithms in order to generate test programs for the automatic testing of programming languages and domain specific languages. For this purpose, the grammar of the language is interpreted as a meta-model such that programs of the language can be seen as models conforming to this meta-model. As a result, model finders act as program finders that not only ensure the embedded constraints of the language (by means of integrity constraints in the meta-model) but also additional constraints that should hold for the generated test program in mind. Using model finders, many constraints can be posed in order to generate test programs in comparison to random test generators in which the generator's implementation needs to be adjusted to support new constraints. An experimental evaluation demonstrates that the approach is feasible although formal methods are used under the hood.

In future work we want to concentrate on a better integration into the work flow, e.g. by implementing the whole algorithm as an Eclipse plugin. Furthermore, it would be interesting to explore how scalable the approach performs when generating very large instances and determine common patterns in the constraints leading to a possible domain specific constraint language for the generation of test programs.

References

[1] Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge, MA, USA (April 2006)

[2] Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., Drechsler, R.: Verifying UML/OCL models using Boolean satisfiability. In: Design, Automation and Test in Europe. (March 2010) 1341–1344

[3] Cabot, J., Clarisó, R., Guerra, E., de Lara, J.: Verification and validation of declarative model-to-model transformations through invariants. Journal of Systems and Software **83**(2) (2010) 283–302

[4] Object Management Group: OMG Meta Object Facility (MOF) Core Specification. (August 2011) Version 2.4.1.

[5] Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: ACM Conf. on Programming Language Design and Implementation. (June 2011) 283–294

[6] Cuoq, P., Monate, B., Pacalet, A., Prevosto, V., Regehr, J., Yakobowski, B., Yang, X.: Testing Static Analyzers with Randomly Generated Programs. In: Int'l Symposium on NASA Formal Methods. (April 2012) 120–125

[7] Boyapati, C., Khurshid, S., Marinov, D.: Korat: automated testing based on Java predicates. In: Int'l Symp. on Software Testing and Analysis. (July 2002) 123–133

[8] Khurshid, S., Marinov, D.: TestEra: Specification-Based Testing of Java Programs Using SAT. Journal of Automated Software Engineering **11**(4) (October 2004) 403–434

[9] Godefroid, P., Kiezun, A., Levin, M.Y.: Grammar-based whitebox fuzzing. In: ACM Conf. on Programming Language Design and Implementation. (June 2008) 206–215

[10] Majumdar, R., Xu, R.G.: Directed test generation using symbolic grammars. In: Int'l Conf. on Automated Software Engineering. (November 2007) 134–143

[11] Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework. 2 edn. Addison-Wesley Professional, Amsterdam (December 2008)

[12] The Eclipse Project: Xtext http://www.eclipse.org/xtext.

[13] Parr, T.: The Definite ANTLR4 Reference. The Pragmatic Bookshelf (September 2012) First beta release.

[14] Warmer, J., Kleppe, A.: The Object Constraint Language: Precise modeling with UML. Addison-Wesley Longman, Boston, MA, USA (March 1999)

[15] The Eclipse Project: OCLinEcore http://wiki.eclipse.org/MDT/OCLinEcore.

[16] Berardi, D., Calvanese, D., Giacomo, G.D.: Reasoning on UML class diagrams. Artificial Intelligence **168**(1-2) (October 2005) 70–118

[17] Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: A Challenging Model Transformation. In: Int'l Conf. on Model Driven Engineering Languages and Systems. (October 2007) 436–450

[18] Cabot, J., Clarisó, R., Riera, D.: Verification of UML/OCL Class Diagrams using Constraint Programming. In: IEEE Int'l. Conf. on Software Testing Verification and Validation Workshop. (April 2008) 73–80

[19] Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language reference manual. Addison-Wesley Longman, Essex, UK (January 1999)

[20] Pérez, C.A.C., Buettner, F., Clarisó, R., Cabot, J.: EMFtoCSP: A Tool for the Lightweight Verification of EMF Models. In: Formal Methods in Software Engineering: Rigorous and Agile Approaches. (June 2012)

[21] de Moura, L.M., Bjørner, N.: Z3: An Efficient SMT Solver. In Ramakrishnan, C.R., Rehof, J., eds.: Tools and Algorithms for Construction and Analysis of Systems. Volume 4963 of Lecture Notes in Computer Science., Springer (April 2008) 337–340

[22] Schmideg, A.: CoffeeScript plugin for Eclipse using Xtext https://github.com/adamschmideg/coffeescript-eclipse.

[23] Gerlach, S.: Xtext based Lua-Editor Blog entry at http://xtexterience.wordpress.com/2011/05/17/xtext-based-lua-editor/.

[24] Seignard, X.: BibTEX Xtext https://github.com/xseignard/bibtex-xtext.