Static analysis and testing of executable DSL specification

Qinan Lai[1], Andy Carpenter[1]
[1]*School of Computer Science, the University of Manchester, Manchester, UK*
{laiq,afc}@cs.man.ac.uk

Keywords: ALF; fUML; DSL; modelling language; behavioural semantics; static check

Abstract: In model-driven software engineering, the syntax of a modelling language is defined as a meta-model, and its semantics is defined by some other formal languages. As the languages for defining syntax and semantics comes from different technology space, maintaining the correctness and consistency of a language specification is a challenging topic. Technologies on formal methods or sophisticated dynamic verification have been developed to verify a language specification. While these works are valuable, they can be hard to apply to a complex language in reality. In this paper, extended static checking and testing are used to maintain the correctness of a language specification, and the techniques are applied to a case study that formalises WS-BPEL to a model-based specification defined by OMG standard fUML and ALF. Several categories of different errors are identified which can happen during semantics development, and how our framework can simplify the checking on them by static checking and direct testing of executable models is discussed.

1. INTRODUCTION

It is common to need to create a new Domain Specific Language (DSL) and a set of supporting tools. Model-driven technologies address several aspects of the development of a DSL, for example EMF/Xtext [3]/GMF support syntax development and OCL allows the definition of static semantics. Experience has showed that basing tool development on model-driven technologies is simpler and faster than traditional language parser/compiler or interpreter approaches. Recently researchers have sought ways to extend the use model-driven technologies to definition of the behavioural semantics of a DSL [6, 1, 8]. .

In practice, a DSL specification usually defines its syntax and semantics in an abstract way. Tools to support a DSL are built by implementing an interpretation of this specification. Compared to other means of defining DSL specification, a model-based description has the advantage that it is both human understandable and machine processable. By exploiting the generation aspects of model-driven engineering tooling implementations can be created directly from the specification eliminating the possibility of interpretation errors.

However, even when using a model-based approach, the different aspects of a DSL defined separately against independent meta-models. This means that there is not a tool that can consider all aspects of the DSL specification and identify, for example, inconsistencies between them or errors in embedded specifications. This is a known source of errors; for example for many years the OCL constraints embedded in the UML superstructure specification contained more than a hundred syntax errors [9], which were eventually removed in UML 2.4 beta version.

In this paper, a unified and formalised definition of the Business Process Execution Language DSL is created using the Action Language for fUML (ALF) [5]. This specification forms a case study that is used to identify the kinds of errors can happen while creating a DSL definition. From the types of errors static checks to programmatically identify errors are being developed. The aim is to exploit the unified description to reduce the effort needed to create error free DSL tooling.

The contributions of the paper are: (1) Identification of the seven categories of common error patterns that can appear in a DSL specification; these patterns are introduced in section 2. It is identified that most of them are small errors, and simple automatic technologies can counter them. (2)

An extensible framework that performs static analysis and testing of a DSL specification is proposed to check these errors. The framework uses an extended ALF language [5] to compose a DSL; currently it could check inconsistency/syntax and many bad practices on a DSL specification. The framework also supports to generate an EMF based DSL interpreter prototype, which could be used to test logic and runtime errors.

2. ERRORS AND BAD PRACTISES IN DSL SPECIFICATION

In this section, the context of how the errors are identified is given. The WS-BPEL language is a DSL aimed for web service composition. Its syntax is defined as XML, and its semantics is defined by natural language, but several works tried to formalise it [4]. We tried to formalise it to a model-based specification, which means creating a MOF based meta-model, formalising the well-formed rules to OCL and modelling the behavioural semantics of an operational language. The framework of defining the BPEL specification is based on our previous work [6]. The meta-model, the OCL constraints and the behaviours are all defined as ALF programs.

The meta-model of BPEL is created by translating the Ecore model of BPEL from Eclipse BPEL designer project to ALF structures.

Figure 1 introduces how our framework could specify, testing and statically check a complete DSL specification. Firstly the BPEL meta-model is defined as an ALF program. ALF syntax for UML units modelling captures the meta-model, and the ALF statements and expressions captures the behavioural semantics. The ALF program is defined in an Xtext-based editor, which also provides static checkers which could report checkable errors to the DSL designer. By testing the ALF program through a generated EMF application, new errors could be found, and new static checkers could be created and integrated to the framework with minimal effort.

2.1 Build ALF executor as a code generator

The ALF open source implementation can directly execute ALF programs. However in our experience, the software is not easy to use. Firstly it does not support some necessary concepts, such as inheritance of signal receptions and operation overloading. Secondly the error message given by it is not clear enough, for example, for many different types of errors it will always report internal reference errors. Practically we try to execute the DSL specification by transforming the ALF based spec to an Ecore model, and directly map the operation body to Java code which embedded as the Genmodel annotations. Thus EMF will generate a Java application that has a one-to-one mapping to

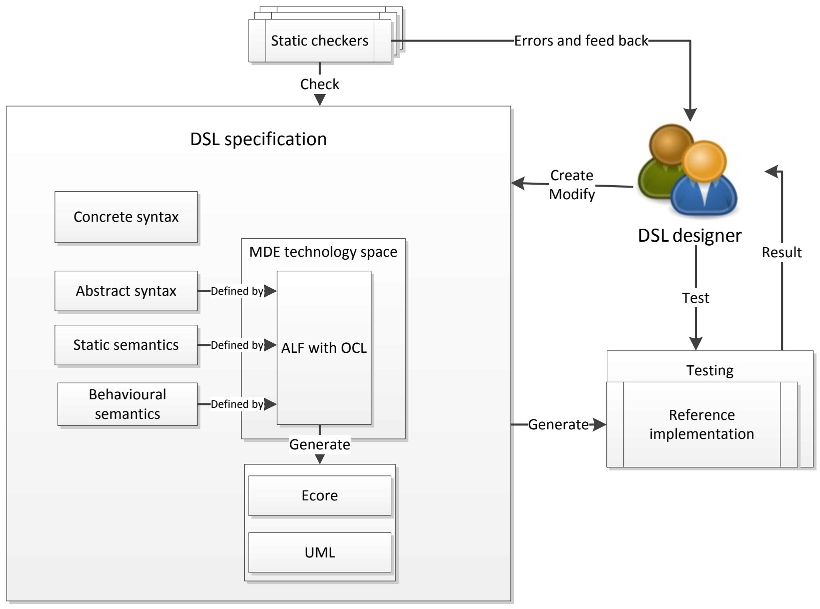


Figure 1 Framework overview

the concepts defined in the specification. Some concepts that do not have a one to one mapping are tried to generate semantically similar code, for example, translating ALF active class as a class implements `Runnable` interface, and start an active class become creating a thread.

By generating Ecore model with Genmodel annotations, the model editor can be reused to create DSL testing models. As a result, a prototype of DSL interpreter and code editor based on EMF are generated from the DSL specification.

2.2 Identifying errors by testing executable DSL specification

In this process of defining BPEL, the errors met were documented. The process of error identification and the creation of static checkers work as below.

Firstly, while developing the ALF programs, test cases were created and the ALF programs are tested by testing the generated EMF application. In this process, many types of error could happen. The code generator could generate wrong code, or there could be errors in the ALF program. The errors that happened in the ALF program, in other words, the BPEL specification was relevant to this paper. Once such an error was identified, they were documented. And then the source and reason of the error was analysed. Finally, static checkers were created and dropped to the ALF editor, so the same types of error would be eliminated or reduced.

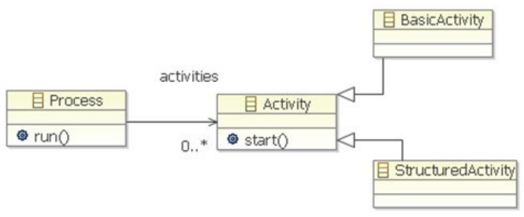
It was identified that these errors were easily introduced. If there was no static checker, they would happen again and again. In summary, 32 error patterns are identified, and they can be categorised as the following 7 kinds of errors. The principle of the categorisation was based on the source and the reason of the errors.

2.3 Errors identified in BPEL case study

The **syntax error** is the most common kind of error. It includes wrong syntax, type mismatch and any violations on the well-formedness of the modelling languages. Despite they are not hard to check, due to the fact that behavioural semantics are defined in another technology space, the tools that can take all the kinds of errors into consideration is not valid.

Inconsistency errors can happen between the definitions of different aspects of a DSL. The first type of consistency is horizontal consistency, which can happen when the meta-model, the static semantics and the behavioural semantics referred to an invalid concept. Vertical inconsistency may happen when the meta-model changed, but the model that conforms to the meta-model does not change. Both horizontal and vertical inconsistency can easily happen when the DSL specification evolves. A small rename of one class in the meta-model can cause all the semantics models that referred to that model become invalid.

The example in Figure 2 shows an example of



```
Context Process
inv: activities->size()>0
```

```
Process::run(){
  for (Activity act: this.activities){
    act.run();
  }
}
```

Figure 2 Inconsistency example

syntax and inconsistency error. The OCL invariance called activity property, but in the meta-model it is called activities. In the behavioural specification, the `run()` operation is invoked, however in the meta-model such an operation is undefined.

Conflicting errors can happen in static semantics definition, where invariance on the meta-model conflicting with each other. It can also happen if the pre- and post-conditions of an operation is conflicting with the static semantics. It is also possible that the invariance of the meta-model conflicts that leads to an unsatisfiable model, which means there is no model which could be instantiate that conforms to the meta-model.

Deficiency can happen when the DSL specification lacks some certain properties. One common category of bad practice is unused concepts or undefined operation stubs. Another deficiency error is signal deficiency, which could happen in the behavioural semantics definition when the active class and signal models are used. Consider the example ALF code:

```
public active class Execution {
  public receive signal SignalStart{}
}do{
   accept(SignalStart){
        //do something
    }
}
```

When the class Execution is instantiated, it will wait for other objects to send a `SignalStart` then it will continue. If this signal is not sent, the active object goes to deadlock due to lack of signal.

Extended static errors are defined as the errors that can be checked by static analysis, but they do not belong to the syntax. In fact, many bad code practices and errors belong to this category. For example:

Comparing multiple valued variable with null

```
if (structuedActivity.activities==null)
```

should be

```
if(strucutedActivity.activities
  ->isEmpty())
```

The "`instanceof` expression always return false" is another example, take the same meta-model in Figure 1, and assume that `process` is an instance of `Process`

```
if (process instanceof Activity){}
```

the condition of the if statement will always remain as false.

These kinds of errors are usually platform-specific to ALF language. However, considering the action languages for behavioural modelling share some common design principles and even syntax are similar. They are usually able to direct manipulating models, have higher abstraction level and support OCL-like syntax, the principles of static errors can be adapted to other languages.

Platform specific errors can happen when the developers wish to use the DSL standard as platform independent models, and generate platform specific models from it. For example, if the developers want to generate a Java-based interpreter of the DSL, the DSL models must avoid names preserved in Java. If the model in Figure 2 is used to generate Java code, it will override `java.lang.Process` class and result compiling error.

Another example is to enforce the naming rules of Java. Any string could be legal names in ALF, however, this lead to compile errors or code that are hard to understand.

Logic errors and runtime errors can still happen, and they are easy to identify by testing rather than static checking.

3. STATIC ANALYSIS AND TESTING ON DSL SPECIFICATION

These errors identified in section 2 should be avoided by some automatic technology, and when developing a DSL specification, the developers should apply automatic checking technology. We designed a framework and which could specify a complete DSL specification, and then perform static analysis of the semantics specification to check syntax error, inconsistency error and other static errors. Logic errors are also testable by directly execute the specification. Our framework uses the syntax of ALF language plus adding OCL annotations to it. The meta-model is defined as ALF units. The OCL constraints are specified as an

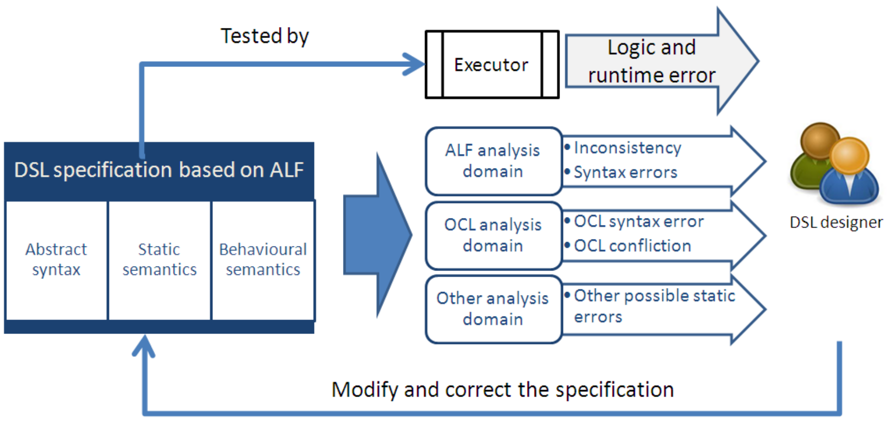


Figure 3 Error checking and testing

annotation. The behavioural semantics are defined as activities and operations. The framework of specification and analysis is developed using Xtext.

The architecture of the static analysis is listed in Figure 3. Different kinds of errors can be checked by integrating relevant analysing technology. Because the specification is defined by ALF language, errors can naturally checked by Xtext validators.

In such a specification, abstract syntax, static semantics and behavioural semantics are defined in a single model-driven technical space. Unlike defining them in different technical spaces that are hard to check the consistency, the syntax errors and inconsistency errors can be easily detected. The detection of inconsistency and syntax errors become the same problem of checking the validity of ALF programs. By defining the grammar of ALF and resolving the internal references, Xtext can report syntax and inconsistency errors while editing the ALF program.

The Xtext validator will check the errors that are checkable in ALF domain. By using the extension points of EMF plugin, it is possible to integrate other types of validators. Currently the framework supports to invoke OCL validator, other validators are still under development.

The Xtext validator works as below: syntax errors can appear in ALF text or OCL text. Xtext will automatically check the errors which could be checked by the parser. A type system is developed to check type errors in the expressions. Separate validator rules are defined to check well-formedness rules, for example, an operation with a return type must have a return statement in the entire execution path. OCL syntax errors are checked by invoking OCL validator in EMF. Extended static errors and platform specific errors can be checked by the same principle. All the static checkers require tens to hundreds of lines of code, which are not hard to create, but it showed that the checkers could significantly reduce the errors in the specification.

Most logical and runtime errors are hard if not possible to check by static analysis technologies. However, some particular kinds of runtime errors can be checker, for example, null pointer dereference, impossible or redundant type cast.

4. FURTHERWORK

There are several unfinished works. There are still some static checkers that are under research. Conflicting errors are not directly checkable by Xtext validation rules. One possible way to check it is to translate the DSL spec to another analysis domain and map the analysis result back to the users. UMLtoCSP [2] is a tool which can check OCL conflicts. Currently we are working on how to use this tool to report conflict errors. Because this process contains translations, how to back annotate the error message produced by the analysis domain to the definition domain remains to be researched.

Some Deficiency errors such as unused models or empty stubs can be easily checked by our framework. Currently our approach for checking signal deficiency is a lightweight approach, which only report error when one active class accepts some signal, but there is no object that has sent these signals. We wish to seek other ways that can check more complex cases.

Currently the generation of DSL interpreter does not support all the concepts defined in ALF standard, it does not support direct use of OCL-like expressions, the code still need some manual work to test. It is interested to fully automate the generation of an interpreter with no limitations.

The framework to use a unified definition to define, check and test a DSL specification is only tested by one case study. The behavioural semantics is based on the imperative paradigm. It is necessary to test whether the same technique can be applied to declarative languages, because there is a large number of DSLs which are declarative languages. It is planned that to carry out another case study for creating a model-based specification for a small functional programming language.

5. CONCLUSION

In this paper, the correctness issue of a DSL specification has been discussed. Seven categories of error that can occur during the development of a specification have been identified and introduced. It has been demonstrated that most of these errors can be detected using a simple static checker, making their removal from specifications a trivial task. The use of generating an implementation from a specification has also been described. This has the advantage of eliminating interpretation errors from the process of creating DSL tooling. Finally an extensible framework that brings together the integration of static checks and the generation of implementations has been outlined.

REFERENCE

[1] Lionel Briand, Clay Williams, Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. *Weaving Executability into Object-Oriented Meta-languages*, volume 3713 of *Lecture Notes in Computer Science*, pages 264–278. Springer Berlin / Heidelberg, 2005.

[2]Jordi Cabot, Robert Clarisó, and Daniel Riera. Umltocsp: a tool for the formal verification of uml/ocl models using constraint programming. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE ’07, pages 547–548, New York, NY, USA, 2007. ACM.

[3] S. Efftinge and M. Völter. oaw Xtext: A framework for textual dsls. In *Workshop on Modeling Symposium at Eclipse Summit*, volume 32, 2006.

[4] D. Fahland and W. Reisig. Asm-based semantics for bpel: The negative control flow. In *Proc. 12th International Workshop on Abstract State Machines*, pages 131–151. Citeseer, 2005.

[5] Object Management Group. Action language for foundational uml (alf) 1.0 - beta 1. www.omg.org/spec/ALF/, 2010.

[6] Qinan Lai and Andy Carpenter. Defining and verifying behaviour of domain specific language with fuml. In *Proceedings of the Fourth Workshop on Behaviour Modelling - Foundations and Applications*, BM-FA ’12, pages 1:1–1:7, New York, NY, USA, 2012. ACM.

[7] Andreas Prinz, Markus Scheidgen, and Merete Tveit. A model-based standard for sdl. In Emmanuel Gaudin, Elie Najm, and Rick Reed, editors, *SDL 2007: Design for Dependable Systems*, volume 4745 of *Lecture Notes in Computer Science*, pages 1–18. Springer Berlin / Heidelberg, 2007.

[8] Markus Scheidgen and Joachim Fischer. Human comprehensible and machine processable specifications of operational semantics. In *Proceedings of the 3rd European conference on Model driven architecture-foundations and applications*, ECMDA-FA’07, pages 157–171, Berlin, Heidelberg, 2007. Springer-Verlag.

[9] C. Wilke and B. Demuth. Uml is still inconsistent! how to improve ocl constraints in the uml 2.3 superstructure. *Electronic Communications of the EASST*, 44, 2011.