Verification and validation of domain specific languages using Alloy

Ana Garis[1] and Alejandro Sánchez[1]

Universidad Nacional de San Luis, Argentina
{agaris,asanchez}@unsl.edu.ar

Abstract. A *domain specific language* (DSL) focuses on a particular problem domain, facilitating the specification of its instances. Since they are frequently defined using imprecise languages, such as UML, they present ambiguities and their *verification and validation* (V&V) becomes complex. This paper proposes an approach to the precise definition of DSLs using Alloy – a formal language with tool-support that enables its V&V. The approach is illustrated with a DSL for Software Architecture.

1 Introduction

A *domain specific language* (DSL) [18] restricts its primitives to a problem domain, usually aiming at facilitating the development of specifications by domain experts. DSLs are typically defined through a metamodel that includes the language elements and the relationships among them [17]. Usually, *UML class diagrams* (CD) [11] are used for this [3], with the main disadvantage of an imprecise semantics, and the lack of a tool-supported mechanism for checking the quality of the resulting metamodel.

This paper proposes an approach that aims at addressing such disadvantage. The approach uses Alloy, a formal language designed for performing automatic analysis, and whose models resemble class diagrams [9]. Alloy includes friendly tool-support for V&V, which is based on a bounded SAT analyser. The mechanism assumes a metamodel specified as a CD, that can optionally be enriched with formulas expressed in the *Object Constraint Language* (OCL) [12]. It prescribes translating the CD and associated OCL formulas into Alloy, verifying and validating the model in Alloy's framework, and then translating it back into a CD and OCL formulas. Domain experts iterate improving the model's quality using Alloy. Therefore, it is possible to generate instances of the model, establish if it is inconsistent, find a counterexample for some assertion, study generated instances and modify the model accordingly, extend it with further formulas that need to be verified, and generate instances again.

A DSL for Software Architecture called Archery [14, 15] is used to illustrate the approach. The language is designed for modelling, animating, analysing, and verifying software architectures in terms of architectural patterns. The language semantics are given by an encoding into a process algebra, but its metamodel is described using a CD, which permits specifying ill-defined models. We study Archery's metamodel and suggest improvements.

The work is framed within the approach described in [7]. The translations the approach relies on were prototyped in Haskell [8] and are available at [13].

Although, we sketched the proposed approach in [7], it omitted the required specifications to be applied in particular domains. This work details how use it for V&V of DSLs.

The contribution of the paper lies in a tool-supported approach based on Alloy to the V&V of DSL metamodels and in the development of an example in the domain of Software Architecture.The rest of the paper is structured as follows: sections 2 and 3 briefly describe the Archery language and Alloy's framework, respectively; section 4 describes how models in Alloy are obtained from CDs; section 5 illustrates the approach by performing the V&V of Archery's metamodel; section 6 describes related work; and section 7 concludes the publication.

2 The Archery language

The Archery language is for modelling the structure and behaviour of software architectures. It allows defining the basic building blocks of architectures: (architectural) patterns and (architectural) elements. A pattern consists of a set of elements representing either component or connector types, that are specified in terms of their interfaces and behaviours. Interfaces are sets of ports, atomic events of interaction, and behaviours are sequential processes that describe how the activities in element's instances take place. The client-server pattern, and the pipes and filters pattern, for instance, are shown in Listing 1.

```
pattern ClientServer()
element Server()
  interface in rreq; out sres; act cres;
  proc Server() = rreq.cres.sres.Server();
element Client()
  interface in rres; out sreq; act prcs;
  proc Client() = prcs.sreq.rres.Client();
end
pattern PipeFilter()
element Pipe()
  interface in acc; out fwd;
  proc Pipe() = acc.fwd.Pipe();
element Filter()
  interface in rec; out snd; act trans;
  proc Filter() = rec.trans.snd.Filter();
end
```

Listing 1. Client-server and pipes and filters patterns

Architectures are built out of defined patterns and elements. An architecture is regarded as a pattern instance that describes a particular configuration of element instances through a set of attachments linking their ports, and a set of renamings changing the externally visible names of ports. Both patterns and elements act as types for instances, which are kept and referenced through typed variables. A variable has an identifier and a type that must match an element or pattern name. Allowed values are instances of a type, that do not necessarily need to match the variable's own type. An attachment includes a port reference to an output port and a port reference to an input port. A port reference is a pair of identifiers that identify a variable, and a port of the variable's instance. In addition, the language supports hierarchical composition by allowing the definition of configurations where attachments indifferently link ports of pattern and element instances. For example, the architecture in Listing 2 defines a server, which is hierarchically composed of instances of the pipes and filters pattern.

```
cs : ClientServer = architecture ClientServer()
instances
  s1 : Server = architecture PipeFilter()
  instances
    f1:Filter=Filter(); p1:Pipe=Pipe(); f2:Filter=Filter();
  attachments
    from f1.send to p1.accept;
    from p1.forward to f2.rec;
  interface
    f1.rec as rreq;
    f2.send as sres;
  end
end
```

Listing 2. A hierarchically composed server

The CD specifying the ARCHERY's metamodel for architectures is shown in Figure 1. It is taken from [15] with the exception of `PortType`. A class specifying the distinction the language makes among ports was omitted in the original CD, but it is included at this stage to enable a richer analysis.

The metamodel is underspecified and presents several issues. We illustrate the approach by making evident and addressing some of them.

3 Alloy

In Alloy, a *signature declaration* denotes a set of atoms. An *atom* is a unity with three basic properties: it is indivisible (it cannot be divided into smaller parts), it is immutable (its properties remain over time), and it is uninterpreted (it does not have inherent properties). Signature declarations can introduce *fields*, which represent a relation among signatures. Listing 3 shows the signature declarations that constitute ARCHERY's metamodel in Alloy.

```
sig Instance { prt: some Port }
sig ElementInstance extends Instance {
   act: set Act }
sig PatternInstance extends Instance {
   att: set Attachment,
   ren: set Renaming,
```

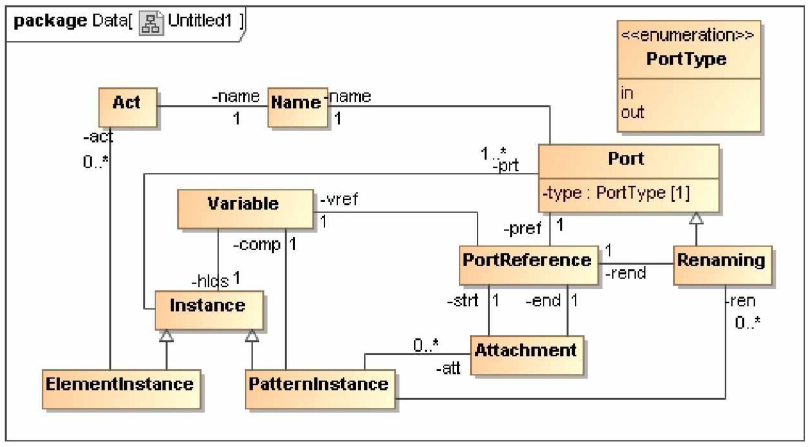

Fig. 1. Class diagram for Archery's architectural specifications

```
    comp: some Variable }
sig Attachment {
    strt: one PortReference,
    end:  one PortReference }
sig PortReference {
    pref: one Port,
    vref: one Variable }
sig Port {
    name: one Name,
    type: one TypePort }
sig Renaming extends Port {
    rend: one PortReference }
sig Variable { hlds: one Instance }
sig Act { name: one Name }
sig Name{}
enum TypePort { In, Out }
```

Listing 3. Metamodel of Archery's architectures in Alloy

The cardinality in relationship between a signature and another can be constrained using the keywords as follows: **lone** for zero or one, **one** for exactly one, **some** for one or more, and **set** for any number.

Facts, *predicates* and *functions* describe invariants, named constraints, and named expressions, respectively. The difference between a fact and a predicate is that the former always holds, while the latter is only verified when invoked. Invariants can also be defined in the context of each signature. *Assertions* allows to express properties that are expected to hold as consequence of specified facts.

The analyser is instructed through *commands* **run** and **check**. Command **run** checks model consistency by requesting a valid instance, and command

check verifies an assertion by searching for a counterexample. Both commands optionally define a scope, overriding the default bound of the number of instances allowed for each signature.

Since in Alloy everything is a relation, it defines the typical set's relational operations: `+` (union), `-` (difference), `&` (intersection), `.` (join), `->` (cartesian product). It also provides two important operators over binary relations, that make its logic more expressive than first-order logic: `^` (transitive closure) and `*` (transitive-reflexive closure).

4 Modelling domain specific languages in Alloy

The translations into Alloy and back are organised in four modules [5]. Prototype tools `CD2Alloy` and `OCL2Alloy` accept a CD and an OCL specification, respectively, and yield the corresponding Alloy model. The dual prototype tools are `Alloy2CD` and `Alloy2OCL`, which accept an Alloy model, and produce a CD and a OCL specification, correspondingly. CD and OCL specifications are handled using the OMG standard XML Metadata Interchange (XMI) format. Listing 3 depicts the model generated from the CD corresponding to the metamodel of Archery's architectures.

A signature is defined for each class in the CD as follows: `ElementInstance`, `PatternInstance`, `Instance`, `Attachment`, `PortReference`, `Renaming`, `Variable`, `Port`, `Act`, `Name` and the class enumeration `TypePort`. Note the inheritance relation is represented in Alloy using keyword **extends** and abstract classes are marked with keyword **abstract**.

Associations corresponds to fields. Signature `Instance`, for example, declares a field `prt` and signature `ElementInstance` a field `act`. The multiplicity in association ends, also has an equivalence in Alloy: `0..*` is **set**; `1..*` is **some**; `0..1` is **lone**, `1` is **one**; and if it is absent, the default is **set**. Therefore, keyword **some** in the declaration of field `prt`, for instance, indicates that each `Instance` has one or more ports.

5 Verifying and validating domain specific languages

The V&V of the DSL's metamodel takes place once its Alloy model is obtained. It is performed as iterations in which domain experts generate instances, modify the model, and generate instances again to confirm consequences of changes. Modifications can be to correct relations, or to add formulas that forbid incorrect instances. Instances are generated either to show examples of the model, or to show counterexamples that disproof a given formula.

It is suggested to start with the generation of examples, allowing domain experts to gain a better understanding of the model. This is done with the **run** command, which in its most simple variant (**run** {}), is usually enough to simulate and find problems or underspecifications in the model, namely, the model allowing instances that are forbidden in the domain.

Subsequently, the model is extended with formulas removing incorrect instances. Such formulas are checked with **check** commands, which generate counterexamples when they are not verified, and allow domain experts to learn where a given formula is weak.

These steps are illustrated with the Alloy model of ARCHERY's metamodel. Figure 2 shows an instance generated with the analiser that is enough to find several problems. The instance shows:

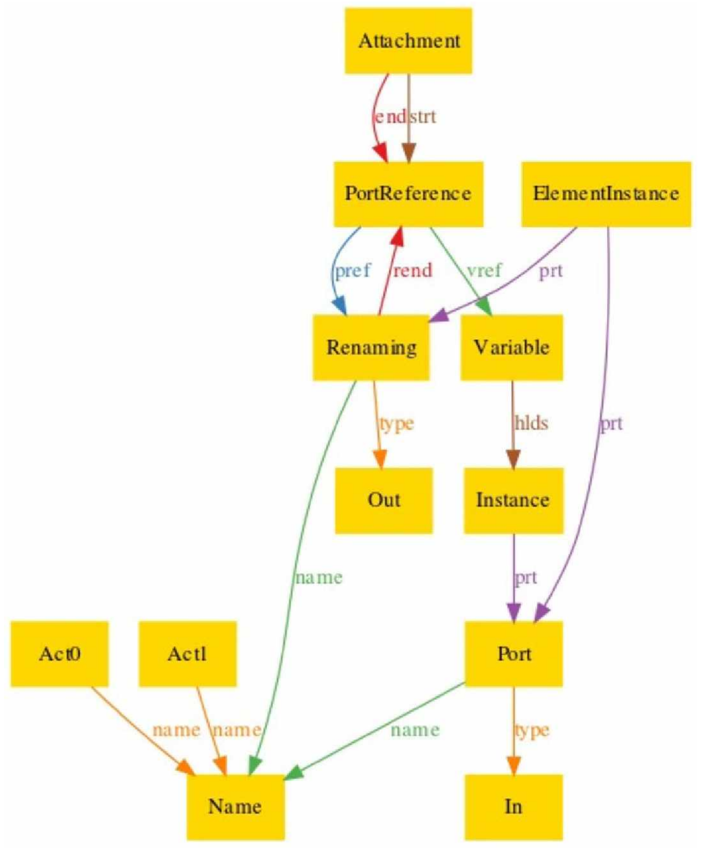

Fig. 2. An instance found with Alloy analyzer.

(i) An instance of class `Instance` that is a superclass of `ElementInstance` and `PatternInstance`. This class is an abstraction of element and pattern instance, which represent components and architectures, respectively. Direct occurrences of this abstraction are incorrect.

(ii) Actions (`Act0` and `Act1`) unrelated to an element instance. All actions must be contained in an element instance, since they represent an observable transition in the state of the component.
(iii) A name (`Name`) is shared between ports (`Renaming` and `Port`) that belong to the same element instance. However, names cannot be shared between ports (including renamings) or actions of the same instance.
(iv) An attachment references the same port twice. The fields `strt` and `end` connect `Attachment` to the same `PortReference`. This makes no sense, since a component cannot interact with itself at the same port. Moreover, communication flows from outward ports to inward ports.
(v) A renaming references to itself. A renaming changes the name of a port, and no cycle can be created by the reference it creates.

Other similar detected problems, such as `Attachment` not having an associated pattern instance, are omitted in this description.

We correct problems (i) to (v) by modifying the model and adding constraints to it. Line 1 addresses problem (i) by declaring signature `Instance` as **`abstract`**: only instances of `ElementInstace` or `PatternInstance` are permitted. Problems (ii) to (v) are addressed including facts in lines (2) to (5), one fact in each respective line, making the model stronger.

```
1 abstract sig Instance { prt: some Port }
2 fact { all a: Act | a in ElementInstance.act }
3 fact { all e: ElementInstance | no e.act.name & e.prt.name }
4 fact { all a: Attachment | no a.end & a.strt }
5 fact { all r: Renaming | not r.^(rend.pref) in Renaming }
```

Problem (iv) unveils a more involved issue requiring further treatment. Attachments have a direction that must be respected. We verify if the model ensures such principle by executing a **`check`** command with the assertion as follows,

```
assert attachInOut { all a:Attachment |
  a.strt.pref.type=Out and a.end.pref.type=In }
check attachInOut
```

which requires attachments to connect `Out` ports to `In` ports. The analyser finds the counterexample shown in Figure 3. Attachments connect `PortReference1` to `PortReference0` through fields `strt` and `end`. However, they are referencing to the same out port, thus violating the assertion. Moreover, the counterexample also shows that the essence of problem (iv) was not addressed by the fact in line (4): attachments may include port references that are different, but that refer to the same port. Adding the assertion as a fact addresses both issues.

Iterations continue, as further issues can be observed, until a satisfactory point is reached. Then, the model can be translated back into a CD enriched with OCL. For example, the facts previously presented, are translated (with `Alloy2OCL`) into the OCL specification as follows.

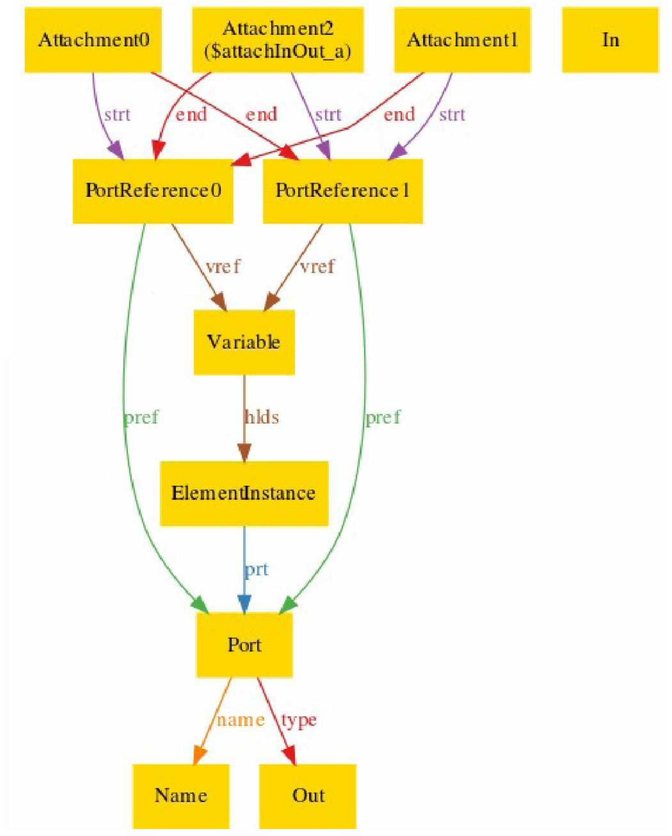

Fig. 3. Counterexample found with Alloy analyzer.

```
context Port inv:
 Instance.allInstances()->exists(v1 | v1.prt->includes(self))
context Act inv:
 ElementInstance.allInstances()->exists(v3 | v3.act->includes(
    self))
context ElementInstance inv:
 (Name.allInstances()->select(v4 | (Act.allInstances()->exists(
    v5 | (self.act->includes(v5) and (v5.oclIsKindOf(Port) and
    v5.oclAsType(Port).name->includes(v4)))) and Port.
    allInstances()->exists(v7 | (self.prt->includes(v7) and v7.
    name->includes(v4)))))->size() = 0)
context Attachment inv:
 (PortReference.allInstances()->select(v9 | (self.end->includes(
    v9) and self.strt->includes(v9)))->size() = 0)
```

6 Related work

Several works, including [1, 2, 10], use formal frameworks to address issues that emerge in the definition of DSLs. Amrani presents a formal specification of Kermeta [1], a metamodelling framework for modeling of DSLs. Its formal specification allows making DSL definitions more precise. Bodeveix et al. combine Bossa (a DSL for process schedulers) and formal method B for ensuring correctness [2]. In particular, B is used to define the correctness of a Bossa specification and to produce certified schedulers. James et al. [10] introduce a methodology for including formal methods in DSLs. The methodology is based on a formal algebraic specification language named CASL. The DSL is first modeled with a CD, then it is automatically translated into a formal specification in CASL.

Alloy potential for DSL modelling has been studied in [16, 4, 6]. Sen et al. expose an approach for using Alloy in order to improve DSLs definition [16]. A set of recommendations are generated to complete partial models which represent DSLs. The completion feature is centred around Alloy. Challenger et al. establish a formal semantics of a DSL for Semantic Web enabled Multi-agent Systems employing Alloy. Static and dynamic aspects of the interaction between agents and semantic web services are considered. Additionally, they explain how to perform automatic analysis for checking these models. Unlike our approach, these works propose the representation of a specific DSL in Alloy, instead of a general approach for modelling DSLs. Alloy is used to support the Lightning tool which allows the representation of modelling languages [6]. The paper describes Lightning's capabilities for verification of a DSL related to structured business processes. This work uses Alloy for supporting the Lightning tool instead of applying directly Alloy's tool-support in order to V&V a DSL metamodel.

7 Conclusion and future work

This paper presented an approach for modelling, verifying and validating domain specific languages (DSLs) using Alloy. It detailed a concrete mechanism for automatically obtaining Alloy models from UML class diagrams (CDs) enriched with formulas specified in the Object Constraint Language (OCL), and a mechanism for translating the models back into CDs enriched with OCL. The approach allows domain experts to gain understanding of their metamodels, modify them, possibly specifying restrictive formulas, so incorrect instances that emerge are prevented. An illustrative example in the domain of Software Architecture was developed.

Future work includes the extension of the approach to study instances of DSLs, and the development of other case studies in order to adjust the approach and validate it further.

References

1. Moussa Amrani. A formal semantics of kermeta. In *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, 2012.

2. Jean-Paul Bodeveix, Mamoun Filali, Julia Lawall, and Gilles Muller. Formal methods meet domain specific languages. In Judi Romijn, Graeme Smith, and Jaco van de Pol, editors, *Integrated Formal Methods*, volume 3771 of *Lecture Notes in Computer Science*, pages 187–206. Springer Berlin Heidelberg, 2005.
3. Achim D. Brucker and Jürgen Doser. Metamodel-based UML notations for domain-specific languages. In Jean Marie Favre, Dragan Gasevic, Ralf Lämmel, and Andreas Winter, editors, *4th International Workshop on Software Language Engineering (ATEM 2007)*. 2007.
4. Moharram Challenger, Sebla Demirkol, Sinem Getir, Marjan Mernik, Geylani Kardas, and Tomaz Kosar. On the use of a domain-specific modeling language in the development of multiagent systems. *Eng. Appl. of AI*, 28:111–141, 2014.
5. Alcino Cunha, Ana Garis, and Daniel Riesco. Translating between Alloy specifications and UML class diagrams annotated with OCL. *Software & Systems Modeling*, pages 1–21, 2013.
6. Loıc Gammaitoni, Pierre Kelsen, and Fabien Mathey. Verifying modelling languages using lightning: a case study. *11th Workshop on Model Driven Engineering, Verification and Validation MoDeVVa 2014*, page 19, 2014.
7. Ana Garis and Alejandro Sanchez. Especificación formal de lenguajes específicos del dominio utilizando Alloy. In *Proceedings of the XVII the Workshop de Investigadores en Ciencias de la Computación*, WICC'15, 2015.
8. Haskell website http://www.haskell.org.
9. Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* MIT Press, revised edition, 2012.
10. Phillip James and Markus Roggenbach. Encapsulating formal methods within domain specific languages: A solution for verifying railway scheme plans. *Mathematics in Computer Science*, 8(1):11–38, 2014.
11. OMG. UML Superstructure, version 2.4.1, 2011.
12. OMG. Object Constraint Language, version 2.4, 2014.
13. Translations MDA - Alloy http://sourceforge.net/projects/alloymda.
14. Alejandro Sanchez, Luis Barbosa, and Daniel Riesco. A language for behavioural modelling of architectural patterns. In *Proceedings of the Third Workshop on Behavioural Modelling*, pages 17–24, 2011.
15. Alejandro Sanchez, Luis S. Barbosa, and Daniel Riesco. Specifying structural constraints of architectural patterns in the archery language. In Theodore E. Simos and Charalambos Tsitouras, editors, *Proceedings of the International Conference on Numerical Analysis and Applied Mathematics 2014 (ICNAAM-2014)*, volume 1648, pages 310008(1) – 310008(5). AIP Proceedings, 3 2015.
16. Sagar Sen, Benoit Baudry, and Hans Vangheluwe. Towards domain-specific model editors with automatic model completion. *Simulation*, 86(2):109–126, 2010.
17. James Willans Tony Clark, Paul Sammut. *Applied metamodelling: A foundation for language driven development.* Ceteva, second edition edition, 2008.
18. Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.