Contextual constraints in configuration languages

Dennis Wagelaar
Vrije Universiteit Brussel
System and Software Engineering Lab
Pleinlaan 2, 1050 Brussels, Belgium
dennis.wagelaar@vub.ac.be

1. Introduction

Configuration languages are a very common solution to manage the variability in software systems. They can take the form of product models for software product lines [6], configuration files for software frameworks[1], workflows for program generators[2] as well as configuration models that are built into the software and can be changed at runtime [3]. Configuration languages can be defined in a variety of ways, ranging from grammars to XML schemas and meta-models. Often, a configuration language defines a number of constraints that rule out any inconsistent configurations. These constraints can be part of the language definition [4] or they can be defined separately [1]. The scope of such constraints is typically limited to so-called *interaction constraints*, which describe what configuration options can be combined with each other. This limit is caused by the vocabulary with which the constraints have to be expressed. This vocabulary is of course only scoped to express the possible configurations.

Another kind of constraint that can apply to configurations, is a *contextual* constraint. Contextual constraints refer to the context in which a software system must work. The relationship between context and programming is described in Context-Oriented Programming [5]. We believe that, in order to describe constraints based on context for a configuration language, there must be an explicit vocabulary of this context. These contextual constraints can then be bound to the configuration language by relating them to the configuration language definition (grammar, schema, meta-model, . . .).

Ontologies have proven to be a suitable format for describing the concepts that can occur in the context [11][8][7]. The standard ontology language OWL [10] provides a way to reason about ontologies with description logic using the OWL DL variant. We have shown in previous work that it is possible to express platform concepts and platform dependency constraints in OWL DL [14], where platform represents a part of the context for a software system. We believe that it is possible to generalise this approach to context and context constraints.

In the rest of this paper, we discuss briefly how context and context constraints can be expressed in OWL DL. We then illustrate how context constraints can be integrated with a configuration language. Finally, we conclude this paper with a summary.

2. Using OWL DL for context

In order to express context constraints, we define an explicit ontology of the context concepts we want to reason about. This ontology is used as a basis to express the current context as well as context constraints. Because the current context and the context constraints use the same context ontology, an automatic inference engine can determine which context constraints are satisfied by the context. We represent context constraints in OWL as classes. As an example of context and context constraints, we will use platforms and platform dependency constraints, respectively. Consider, for example, the "JavaAWTPlatform" platform dependency constraint shown in Fig. 1.

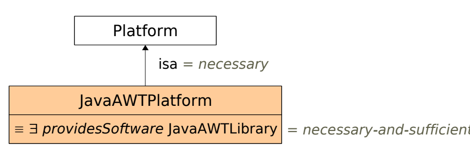

Figure 1. Example platform dependency

"JavaAWTPlatform" is represented as an OWL class with a *necessary* constraint as well as a *necessary-and-sufficient* constraint. Whereas it is *necessary* that each

1 http://www.hibernate.org/hib_docs/reference/en/html/session-configuration.html

2 http://www.openarhictectureware.org/

"JavaAWTPlatform" is a "Platform", being a "Platform" is not *sufficient* for also being a "JavaAWTPlatform". Providing the "JavaAWTLibrary" software, however, is *necessary-and-sufficient* for being a "JavaAWTPlatform". The "JavaAWTPlatform" constraint can be checked against OWL instances that represent platform instances. Each OWL instance – or *individual* – that satisfies the conditions of the "JavaAWTPlatform" class can be considered as an instance of that class. Each platform instance representation that is an instance of a platform dependency constraint, satisfies that platform dependency constraint.

Our platform ontology is not monolithic, but is divided into a network of OWL files. The central part of the platform ontology is made up of several "vocabulary ontology" files, where the word "vocabulary" refers to the fact that the domain concepts are all introduced in these ontologies. These describe the general concept of Platform and its parts. Platform dependency constraints are expressed in terms of this vocabulary ontology, while they are stored in a separate OWL file. This separate OWL file is not considered to be part of the vocabulary ontology, since it expresses only platform dependency constraints rather than platform domain concepts.

Platform instances are also described in a separate OWL file and refer to the same platform vocabulary ontology as the platform dependency constraints. An automatic DL reasoner can be used to verify whether a platform instance satisfies a platform dependency constraint. In addition, DL reasoners can infer a class hierarchy. As we represent context constraints with OWL classes, we can determine which platform dependency constraints are *more specific* than others and hence form a closer match to a platform instance. Translated to context, this allows us to determine the configuration choices that are more specific to the current context.

3. Context constraints in configuration languages

Context constraints can be integrated with a configuration language by binding them to the language definition. The elements that make up a configuration language definition represent the available configuration options. In case of a grammar, each terminal and non-terminal represent a configuration choice. In case of an XML schema, each element represents a configuration choice. In meta-models, the meta-classes represent the configuration choices. We can annotate these elements of the language definition with the corresponding context constraints. Fig. 2 shows how the meta-model of a configuration language for a code generator framework can be annotated with context constraints.

The meta-model is defined using the Eclipse Modeling Framework (EMF) [2]. The EMF meta-modelling language (Ecore) allows for annotations to be added to each meta-element. We have added annotations to each meta-class with a context constraint. Each annotation points to an OWL description of the context constraint. These context constraints can now be used to assist in the configuration process:

- We can adapt the configurator tool to allow for correct configuration choices only, as well as sort the available configuration options *most-specific-first*.
- We can automatically select the best match to the context from a number of pre-defined configurations.
- We can (semi-)automatically generate configurations by using the context constraint class hierarchy and context constraint satisfaction checking.

We use the OWL class hierarchy of the context constraints to determine which configuration, or configuration choice, is more or less specific to the context. To achieve this, a translation must be made from the OWL class hierarchy to an ordered list of configurations, or configuration choices, respectively. A detailed description of this process can be found in [13], section 4.6.1.

4. Discussion

The scope of regular constraints on a configuration language is naturally limited to the vocabulary of that configuration language: the constraints are expressed in terms of the vocabulary that is available. We have identified another kind of constraint that relates to the context of the software system to be configured, which we call context – or contextual – constraints.

We have chosen to represent context and context constraints in OWL DL, which is a standard ontology language. Ontologies have proven to be well-suited for the description of context. Moreover, the fact that the description of context constraints already requires additional vocabulary, justifies the use of a separate formalism. We have shown how our separate description of context constraints can be integrated with the meta-model of a configuration language. We have indicated that this is also possible for other language definition formalisms, such as grammars and schemas. The choice of only annotating the meta-classes in a meta-model – or (non-)terminals in a grammar, or elements in an XML schema – limits the granularity of context constraints. Every instance of that meta-class must introduce the same context constraints, regardless of where they occur in the configuration model. Context constraints on attributes are also not possible, as attributes in a meta-model always have a primitive type: it makes no sense to define context constraints on primitive types. As a result, the configuration language

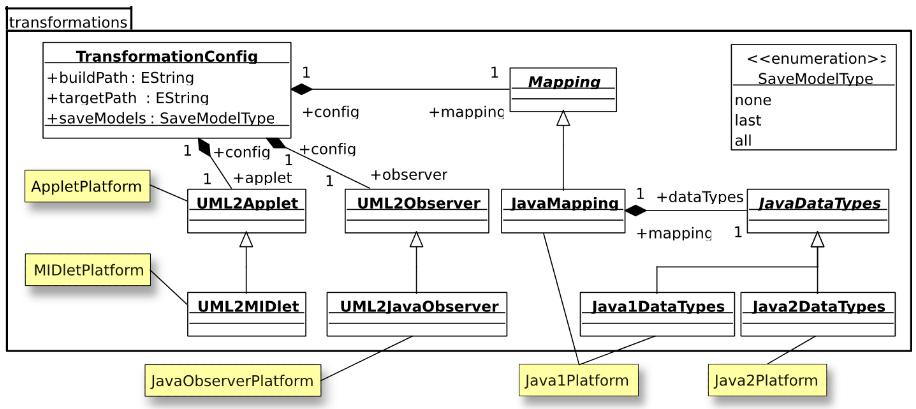

Figure 2. Annotated configuration language for code generator framework

must be designed to allow for annotation with context constraints.

OWL DL performs well in describing *provides/requires*-style constraints, such as context constraints. The automatic hierarchy classification in OWL DL has proven useful to relate context constraints to each other in terms of *specificness*. From all the satisfied context constraints, the more specific constraint is a closer match the context. An important limitation of our method of defining context constraints is the way we chose to combine them: we do a separate satisfaction check for each constraint. A satisfaction check only involves checking that there are instances (context elements) for a context constraint. Sometimes, we want multiple context constraints to be satisfied by the same context element. For example, we require a Java runtime with AWT and we require a Java runtime with the Java 2 Collections API. We generally don't want two separate Java VMs to each satisfy one of the constraints. To solve this problem, the context constraints must refer to each other: "a platform that provides the Java 2 Collections framework and is also a JavaAWTPlatform." However, we don't know which context constraints apply at the time they are defined: only when doing the actual configuring, can we know the applicable context constraints. This means that we cannot define this kind of constraint in OWL DL directly.

Automatic reasoning on OWL DL ontologies is no trivial task: determining standard OWL DL, which corresponds to the $\mathcal{SHOIN}(\mathbf{D})$ description logic, satisfiability has a complexity of NExpTime [12]. When limiting the usage of OWL DL to $\mathcal{SHIF}(\mathbf{D})$, the complexity is reduced to ExpTime [12]. PSpace complexity is only achieved when removing transitive roles (part of $\mathcal{S}$), inverse roles ($\mathcal{I}$) and role hierarchies ($\mathcal{H}$), resulting in the $\mathcal{ALCF}(\mathbf{D})$ description logic [12]. Current DL reasoner implementations can leverage limited usage of OWL DL at least down to $\mathcal{SHF}(\mathbf{D})$, which lacks inverse roles [9]. On the practical side, recent experiments with a $\mathcal{SHF}(\mathbf{D})$ ontology have shown consistent reasoning performance of ¡ 1 second when reasoning about ca. 1000 OWL classes (context vocabulary and constraints), of which about half is completely defined using an equivalence relationship, against ca. 50 OWL individuals (the context). This performance is achieved on a laptop with a 2 GHz Intel Core2 Duo processor. The whole tool setup, including reasoner, can work within 32 MB of RAM for this scenario. This means that we can reasonably expect OWL DL reasoning to scale sufficiently for use on today's desktop- and laptop-class computers and tomorrow's mobile devices. There are scenarios in which run-time adaptation to context with OWL DL context descriptions is feasible. A straightforward example is a software system that will deploy the most appropriate user interface to the end user device that presents itself to the system[3].

References

[1] D. Batory, D. Benavides, and A. Ruiz-Cortéz. Automated analysis of feature models: Challenges ahead. *Communications of the ACM*, 49(12):45–47, Dec. 2006.

[2] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose. *Eclipse Modeling Framework*. The Eclipse Series. Addison Wesley Professional, Aug. 2003.

[3] D. Deridder. *A Concept-Centric Environment for Software Evolution in an Agile Context*. PhD thesis, Vrije Universiteit Brussel, Brussels, Belgium, 2006.

[4] A. v. Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002.

[5] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, Mar. 2008.

[6] C. W. Krueger. New methods in software product line practice. *Communications of the ACM*, 49(12):37–40, Dec. 2006.

3 http://ssel.vub.ac.be/platformkit/instantmessenger/

[7] R. Lewis and J. M. C. Fonseca. *Delivery Context Ontology*. World Wide Web Consortium, Apr. 2008. W3C Working Draft 15 April 2008, [Online] http://www.w3.org/TR/dcontology/.

[8] D. Preuveneers, J. Van den Bergh, D. Wagelaar, A. Georges, P. Rigole, T. Clerckx, Y. Berbers, K. Coninx, V. Jonckers, and K. De Bosschere. Towards an extensible context ontology for ambient intelligence. In P. Markopoulos, B. Eggen, E. H. L. Aarts, and J. L. Crowley, editors, *Proceedings of the Second European Symposium on Ambient Intelligence (EUSAI 2004), Eindhoven, The Netherlands*, volume 3295 of *Lecture Notes in Computer Science*, pages 148–159. Springer-Verlag, Nov. 2004.

[9] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical OWL-DL reasoner. *Journal of Web Semantics*, 5(2):51–53, June 2007.

[10] M. K. Smith, C. Welty, and D. L. McGuinness. *OWL Web Ontology Language Guide*. World Wide Web Consortium, Feb. 2004. W3C Recommendation 10 February 2004, [Online] http://www.w3.org/TR/owl-guide/.

[11] T. Strang, C. Linnhoff-Popien, and K. Frank. CoOL: A context ontology language to enable contextual interoperability. In J.-B. Stefani, I. Dameure, and D. Hagimont, editors, *Proceedings of 4th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems (DAIS2003)*, volume 2893 of *Lecture Notes in Computer Science*, pages 236–247. Springer-Verlag, Nov. 2003.

[12] S. Tobies. *Complexity Results and Practical Algorithms for Logics in Knowledge Representation*. PhD thesis, RWTH Aachen, Aachen, Germany, May 2001.

[13] D. Wagelaar. *Platform Ontologies for the Model-Driven Architecture*. PhD thesis, Vrije Universiteit Brussel, Brussels, Belgium, Apr. 2008.

[14] D. Wagelaar and R. Van Der Straeten. Platform ontologies for the model-driven architecture. *European Journal of Information Systems*, 16(4):362–373, Aug. 2007.