Searching and Using External Types in an Extensible Software Development Environment

Alexander Paar
Department of Computer Science
University of Pretoria
Pretoria 0002, South Africa
alexpaar@acm.org

ABSTRACT

Schema and ontology languages have proved to be useful for conceptualizing knowledge in a variety of applications. In many software projects, XML Schema Definition data types and ontological concept descriptions coexist with programming language class hierarchies. However, only programming language type definitions are fully integrated into today's software development environments. Support for external type systems is spotty. For programmers, it is particularly tedious to search type definitions in XML schema files and OWL ontologies, to browse external type hierarchies, to investigate external type members, and to analyze and comprehend the use of external type definitions in program code. In this work, it will be argued that improved search capabilities are required to ease the use of schema and ontology languages in software projects. Difficulties of searching type definitions in software project workspaces will be indicated. An extensible compiler framework will be outlined that facilitates the use of schema and ontology languages in C# programs. An Eclipse-based integrated development environment will be described that makes XML data types and OWL concept descriptions first-class citizens of the source code editor. Finally, identical search and (just in time) program analysis features for programming language and external type definitions will be suggested.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*data types and structures, frameworks*; D.3.3 [**Programming Languages**]: Processors—*compilers*

General Terms

Integration of programming and ontology languages

Keywords

C#, Zhi#, OWL, XSD

SUITE '10, May 1 2010, Cape Town, South Africa

1. INTRODUCTION

Computers have become fast enough to afford the luxury of using schema and ontology languages such as XML Schema Definition (XSD) [5] and the Web Ontology Language (OWL) [7]. The Extensible Markup Language (XML) [4] and XML Schema Definition have been widely adopted as a format to describe data and to define programming language agnostic data types and content models. Corporations from all sectors have braced to define company specific knowledge using the Web Ontology Language. However, using *external* type definitions (i.e. type definitions based on schema and ontology languages) in software projects is still laborious and error-prone. From the author's experience, this is mainly caused by the deficient integration of external types with the development tools for programming languages. Neither programming language compilers nor integrated development environments are aware of XML data type definitions and OWL concept descriptions. There is no support for 1) searching external type definitions in project workspaces, 2) importing and using external type definitions in program code, and 3) checking the use of external types. It is particularly tedious for programmers to find answers to questions such as "What external types are available in my software project?", "How can external type definitions be used (together)?", and "Where in my program are what external types used?". Program analysis and comprehension are severely limited by a deficient integration of schema and ontology languages with programming languages and software development environments. In particular, the above problems are caused by lacking search capabilities for external type definitions in development tools such as Visual Studio [8] and Eclipse [12]. Using text search utilities such as `grep` is not a solution. XML schema files and RDF/XML [3] representations of OWL ontologies require type system specific logic such as an XML parser and an ontology management system to extract complete information from files. Limited search capabilities are particularly problematic since Singer et al. [11] showed that code search is the most frequent software developer activity.

Unfortunately, XML data types and ontological concept descriptions cannot simply be used in form of programming language proxy classes because of different conceptual bases. In XSD, atomic types can be derived through the application of value space constraints. In OWL, types are inferred based on ontological reasoning. OWL object properties can be used to declare ad hoc relationships between ontological individuals. Ponder the following definitions of an XML data type *age*, OWL concepts *Person*, *Employee*, *Company*

and *Job*, OWL datatype property *hasAge*, and OWL object properties *worksFor* and *hasJob*. Type *age* is defined in XML syntax. The ontology is given in Description Logics [2] notation. Symbol $\top$ denotes the top level concept. In this text, namespaces may be omitted for brevity. The value space of type *age* is restricted to non-negative integer numbers less than 110. The OWL datatype property relates *Person*s with XML *age* values; OWL object properties *worksFor* and *hasJob* relate *Employee*s with *Company*s and *Job*s, respectively. The latter two concepts are declared to be disjoint (i.e. an ontological individual cannot simultaneously be a *Company* and a *Job*).

```
1 <xsd:schema [...]>
2 <xsd:simpleType name="age">
3 <xsd:restriction base="xsd:nonNegativeInteger">
4 <xsd:maxExclusive value="110"/>
5 </xsd:restriction>
6 </xsd:simpleType>
7 </xsd:schema>
```

$$
\begin{array}{l}
1\quad Person \sqsubseteq \top,\ \geq 1 hasAge \sqsubseteq Person,\ \top \sqsubseteq \forall hasAge.age,\\
2\quad \top \sqsubseteq (\leq 1 hasAge),\ Employee \sqsubseteq Person,\ Company \sqsubseteq \top,\\
3\quad \geq 1 worksFor \sqsubseteq Employee,\ \top \sqsubseteq \forall worksFor.Company,\\
4\quad Job \sqsubseteq \neg Company,\ \geq 1 hasJob \sqsubseteq Employee,\ \top \sqsubseteq \forall hasJob.Job
\end{array}
$$

Note that in OWL, in contrast to frame logics, ontological roles form a separate hierarchy of their own. Automatic reasoning is used to compute the class membership of ontological individuals. In contrast to C#, in OWL ontologies, declarations of object property values do not fail immediately. Instead, the reasoner considers the declared piece of knowledge for subsequent deduction steps. If an ontological role relates two individuals then the subject and object of the declared triple are inferred to belong to the domain and range class, respectively. For example, if an ontological individual BILL is related via role *hasAge* with an *age* value, BILL can be inferred to be a *Person*. If it is stated that BILL *worksFor* MICROSOFT one can conclude that BILL is an *Employee* and MICROSOFT is a *Company*. This is only a small contrived example but a plethora of viable ontologies is available on the Web today – encoding valuable knowledge from a variety of domains.

In today's software development environments, however, external type definitions such as ontologies can at best be technically included with the set of files in the project workspace. Still, the semantics of the defined types is concealed in unintelligible text files (containing a bulk of angle brackets in case of XML-based syntax formats). In general, text search tools are insufficient for extracting complete type information from schema and ontology files.

Program analysis and comprehension is limited by the inevitable use of APIs for external type definitions. The code snippet below indicates the instantiation of XML data type *age* by means of the W3C Document Object Model [13]. In the given example, the use of type *age* in C# program code can barely be discovered by program comprehension tools. Automatic debugging techniques such as *delta debugging* [14] regularly find exactly those kinds of errors where external resources that are not subject to built-in static type checking are incorrectly referenced by string literals (e.g., SQL query strings). Search appears to be similarly impaired.

```
1 var pool = new TypePool();
2 pool.LoadSchema(@"Person.xsd");
3 var age = pool.GetSimpleType("age", "ns#");
4 age.Validate(54);
```

The following two integration requirements can be identified. First, schema and ontology languages should be incorporated into a programming language to be amenable to the same search and analysis features that are available for programming language types. Second, software development environments should provide the same search and (just in time) comprehension tools for external and programming language type definitions.

2. THE ZHI# PROGRAMMING LANGUAGE

The Zhi# programming language [10] is a proper superset of ECMA 334 standard C# version 1.0 [6]. The type system of the C# programming language implements *nominal* subtyping. In nominative type systems, type compatibility is determined by explicit declarations. A type is a subtype of another if and only if it is explicitly declared to be so in its definition. The XML Schema Definition type system extends nominal subtyping with *value space-based* subtyping. An atomic data type is a subtype of another if it is explicitly declared to be so in its definition or if its value space (i.e. the set of values for a given data type) is a subset of the value space of the other type. The subset relation of the types' value spaces is sufficient. The two types do not need to be in an explicitly declared derivation path. In the Web Ontology Language, nominal subtyping is augmented by *ontological reasoning*. An inferred class hierarchy can include additional subsumption relations between ontological concept descriptions. Ontological individuals can be explicitly declared to be of a given type and they can be inferred to be in the extension of further concept descriptions based on, for example, particular property values or cardinalities. Some object-oriented programming languages provide a limited set of isomorphic mappings from XML data types to programmatic types. In general, however, compilers for programming languages such as Java or C# are unaware of the subtyping mechanisms that are used for XSD and OWL – and the list of conceivable external type systems and subtyping mechanisms can be arbitrarily extended.

In Zhi#, external types can be included using the keyword *import*, which works analogously for external types like the C# *using* keyword for .NET programming language type definitions. It permits the use of external types in a Zhi# namespace such that, one does not have to qualify the use of a type in that namespace. An *import* directive can be used in all places where a *using* directive is permissible. As shown below, the *import* keyword is followed by a *type system evidence*, which specifies the external type system (i.e. compiler plug-in) that is responsible for the import of the type definitions in the given external namespace. Unlike *using* directives, the alias, which can subsequently be used to represent the external namespace name, is not optional but must be specified. Like *using* directives, *import* directives do not provide access to any namespaces that may be nested in the specified external namespace (based on the namespace scheme of the external type system).

```
import type_system_evidence alias = external_namespace;
```

In Zhi# program text that follows an arbitrary number of *import* directives, external type and property references must be fully qualified using an alias that is bound to the namespace in which the external type is defined. In Zhi#, external type definitions can almost unrestrictedly be used with almost all C# programming language features (since

Zhi#'s support for external types is a language feature and not (yet) a feature of the runtime, similar restrictions to the usage of external types apply as for generic type definitions in the Java programming language). For example, methods can be overridden using external types, user defined operators can have external input and output parameters, and arithmetic and logical expressions can be built up using external objects. Ponder the following code snippet.

```
1 import XML xsd = http://example.org/schema#;
2 import OWL ont = http://example.org/ontology#;
3 class Main {
4  public static void Main() {
5   #ont#Person p = new #ont#Person("Bill");
6   p.#ont#hasAge = 54;
7   p.#ont#worksFor = new #owl#Thing("Microsoft");
8  }
9 }
```

In line 1 and 2, the Zhi# compiler is instructed to parse referenced XML schemas and RDF/XML files and import type definitions of the specified namespaces. Thus, programmers are freed from using additional tools to search schema and ontology files for type and namespace definitions. The Zhi# compiler reports an error if no types are defined in the specified namespaces. The Zhi# IDE described in the following section boasts autocompletion for import statements. In line 5, *Person* Bill is declared in the ontological knowledge base. The Zhi# compiler reports an error if no such concept description exists in the specified namespaces. The Zhi# IDE suggests and autocompletes concept names based on the typed in namespace prefix. In line 6, the power of the "." is used to make a statement about the *age* of Bill. Note how types of different type systems can be used cooperatively in one single statement. Finally, in line 7, it is stated that Bill *worksfor* Microsoft. In OWL, property domain and range restrictions are not authoritative. Instead, the range restriction of role *worksFor* is used at runtime to infer that Microsoft is a *Company*. Based on the domain restriction of *worksFor* one can conclude that *Person* Bill is an *Employee*.

A frequent case that can lead to clashes at runtime is the improper use of disjoint concept descriptions. At first glance, it may be reasonable to state that Bill *worksFor* Chairman, where Chairman shall be a *Job*. However, in the given ontology, the range of role *worksFor* is restricted to *Company*s – and concepts *Company* and *Job* are declared to be disjoint. Taking into account such kind of information does require a significant amount of search that includes manual lookups of property domain and range restrictions and related concept descriptions. This search task is accomplished automatically by the Zhi# compiler. Search results are displayed by the Zhi# IDE. The Zhi# compiler considers search results for type checking. In the following code snippet, an error is reported given the violation of the property range restriction in line 2.

```
1 #ont#Person p = new #ont#Person("Bill");
2 p.#ont#worksFor = new #ont#Job("Chairman");
```

Zhi#'s extensibility with respect to external type systems was achieved by an extensible compiler framework. The Zhi# compiler framework incorporates a full fledged source-to-source compiler for C# version 1.0 [6] plus the syntactical Zhi# language extensions. The Zhi# compiler framework provides the core functionality to type check conventional C# programs and to transform program text into a pretty printed form. The core components were implemented such that, they can use compiler plug-ins for external type systems. For example, the type table of the Zhi# compiler framework can handle method signatures that comprise external type definitions without a priori knowledge about the particular external type systems (e.g., XSD, OWL) that will eventually be used. On the other hand, compiler plug-ins for external type systems can be developed without exhaustive knowledge about the code structure of Zhi# programs. Compiler plug-ins can be provided by implementing two framework extension points for (sub-)typing and program transformation. Plug-ins were implemented for XML Schema Definition and the Web Ontology Language.

3. THE ZHI# IDE

The extensible Zhi# compiler framework lays out the foundation for IDE support for external type systems. The Eclipse-based Zhi# front end supports an `MSBuild`-based build process and facilitates several search tasks that involve external types. Outline views as shown below display XML data types and OWL concept and role descriptions that are contained in the Zhi# project workspace ("What types are available in my software project?"). Source file outline views give an overview of the external types that are actually used in program code ("What types are used in my program?").

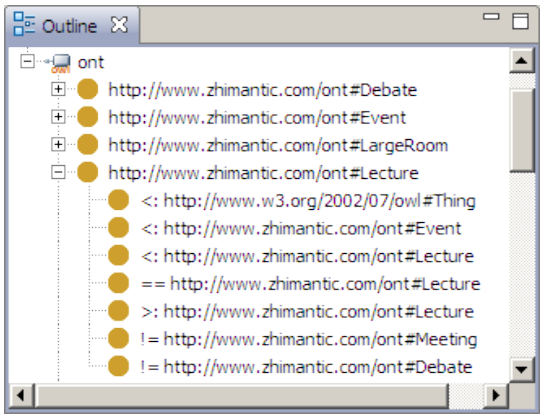

Figure 1: Outline view

The Zhi# source code editor boasts autocompletion for *import* directives and external types based on search results. Developers are freed from searching available namespaces and type definitions in schema and ontology language files and manually filtering and organizing search results.

Autocompletion for *import* directives behaves similarly to search suggestions in search bars. Furthermore, due to the closed world nature of project workspaces the Zhi# IDE is in the position to immediately indicate that no definitions are available in a typed in namespace name.

The integration of external types with the Zhi# programming language makes it possible to filter search results based on the current cursor position in program code. For example, for an ontological host object (i.e. an OWL individual) that is known to be in the extension of a particular class autocompletion may prune ontological roles whose domain restriction is disjoint with the host object class. At the same time, ontological roles whose domain is subsumed by the host object class may be ranked higher in the proposal list

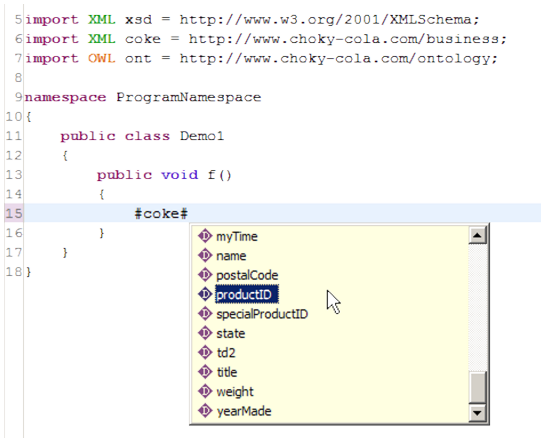

Figure 2: Autocompletion of XML data types

(since it is likely that statements are added to the knowledge base that do not trigger type inference).

Just in time program comprehension is improved in the Zhi# source code editor by tool tips that display metadata about XML data types and OWL concepts and roles when the programmer hovers with the mouse pointer over these elements in program text. Thus, one quickly gets insight into, for example, property domain and range restrictions.

4. CONCLUSION & OUTLOOK

Schema and ontology languages are increasingly used in software development. Searching schema and ontology language files cannot be accomplished with generic text search utilities but requires type system specific tools. In the Zhi# programming language, external type systems are first class citizens. The incorporation of XML data types and ontological concept and role descriptions into a programming language lays the ground for improved search capabilities, which are an integral part of the described integration approach. The Zhi# Code DOM inherently contains structured information about the use of external types in program code. Thus, external types are amenable to the same browsing, searching, and querying techniques that can be used for programming language types. In the Zhi# source code editor, context sensitive search based on the cursor position in program code accomplishes automatic filtering and ordering of search results such as, for example, available ontological roles. As for programming language types, the just in time presentation of search results in form of autocompletion can greatly improve productivity. Given the importance of the source code editor as the programming language front end it is crucial that both the compiler as well as the IDE provide the same level of integration of external types. It is conceivable that the presented approach to extend software development tools with support for external type systems becomes generally accepted. Murphy et al. [9] found out that developers *are* extending their development environments with additional third party tools.

A variety of opportunities exists for further search capabilities based on the integration of schema and ontology languages with a programming language. The Web Ontology Language itself can be used as a query language to retrieve concept and role descriptions of interest (e.g., "All subconcepts of a given concept that contain a property value restriction"). Natural language processing along with standardized [1] label annotations on OWL concepts and roles in form of tagged RDF literals (e.g., "home"@en, "casa"@es) could facilitate search and reuse of domain knowledge in software applications. Improved search capabilities for schema and ontology languages and the incorporation of these formalisms into software development environments are two sides of the same medal that will eventually lead to improved precision and recall of software information systems.

5. ACKNOWLEDGMENTS

The extensible Zhi# compiler framework and the Zhi# IDE were part of the FP6 CHIL project (FP6-506909), partially funded by the European Commission under the Information Society Technology (IST) program. The author acknowledges valuable help and contributions from all partners of the project. Current research is performed under the University of Pretoria Post-doctoral Fellowship Program.

6. REFERENCES

[1] H. Alvestrand. RFC 3066 - Tags for the Identification of Languages. Technical report, Network Working Group, January 2001.

[2] F. Baader, D. Calvanese, D. McGuiness, D. Nardi, and P. F. Patel-Schneider. *The Description Logic Handbook*. Cambridge University Press, Cambridge, United Kingdom, 2003.

[3] D. Beckett and B. McBride. RDF/XML Syntax Specification (Revised). Technical report, World Wide Web Consortium (W3C), February 2004.

[4] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0 (Fourth Edition). Technical report, World Wide Web Consortium (W3C), August 2006.

[5] D. C. Fallside and P. Walmsley. XML Schema Part 0: Primer Second Edition. Technical report, World Wide Web Consortium (W3C), October 2004.

[6] A. Hejlsberg, S. Wiltamuth, and P. Golde. C# Language Specification Version 1.0. Technical report, ECMA International, 2002.

[7] D. L. McGuinness and F. van Harmelen. OWL Web Ontology Language Overview. Technical report, World Wide Web Consortium (W3C), February 2004.

[8] Microsoft Corporation. Microsoft Visual Studio, 2007.

[9] G. C. Murphy, M. Kersten, and L. Findlater. How are Java software developers using the Eclipse IDE? *IEEE Softw.*, 23(4):76–83, 2006.

[10] A. Paar. *Zhi# – Programming Language Inherent Support for Ontologies*. PhD thesis, Universität Karlsruhe (TH), 2010.

[11] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil. An examination of software engineering work practices. In *CASCON '97: Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, page 21. IBM Press, 1997.

[12] The Eclipse Foundation. Eclipse, 2005.

[13] World Wide Web Consortium (W3C). Document Object Model (DOM), 1998.

[14] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, October 2005.