A Test Specification Code Translator Using Ontology and XML/XSLT Technologies

Lim Lian Tze, Tang Enya Kong and Zaharin Yusoff
Computer Aided Translation Unit
School of Computer Sciences
Universiti Sains Malaysia
Penang, Malaysia
{liantze, enyakong, zarin}@cs.usm.my

ABSTRACT

Testing equipment from different vendors utilise different syntax for their test specifications, but writing specifications for the same tests on different platforms is labour-consuming and error-prone. We present a prototype system for the automatic translation of test specifications between platform-specific languages, the framework of which is reusable for similar scenarios or software projects. We first design a generic representation scheme for test specifications. To perform a translation from a source language S to a target language T, a context free language parser module reverse-engineers test specifications in language S to a platform-independent equivalent in the generic syntax, marked up with XML. An XML/XSLT-driven generator module then transforms the platform-independent test specifications to language T syntax. In addition, we encapsulate knowledge about the tester domain in an ontology to facilitate the translation process. We also found ontologies to be helpful tools for requirements gathering, analysis and design.

KEYWORDS

Template-based code generation, domain modelling, ontology, context free grammar, XML, XSLT.

1. Introduction

Testing equipment from different vendors employ different code syntax for developing test programs. This is mainly because vendor-specific tester syntax has been specialised and configured to make full use of vendor-specific tester capabilities.

Writing test specifications for different platforms manually is time and labour consuming, as well as error-prone. We present a prototype translation engine for the automatic translation of test specifications in different platform-specific languages, by approaching the problem as one of code generation and translation. This engine will be able to:

- generate tester-specifications for different vendors based on a generic tester language syntax,
- reverse-engineer vendor-specific tester syntax to generic tester language syntax.

Our solution employs technologies such as the Extensible Markup Language (XML), Extensible Stylesheet Language Transformations (XSLT), ontologies, context free grammar (CFG) and parser generators. The final framework, in part or whole, is reusable for solving problems of a similar nature. In Section 2, we describe the problem briefly, while Section 3 gives a short background on generation in software development. This is followed by a description of the architecture design of our translation engine in Section 4, which is further illustrated with an example in Section 5. Finally, Section 6 and 7 concludes this paper by proposing possible future work and a short summary.

2. Background

Each vendor-specific language has its own syntax. For example, language *A* has a C-like syntax, while language *B* has both Lisp-like and tabular structures. In addition, each language has different names for the test schema types, parameters, and enumeration values. (We refer to these as *test keywords* henceforth.) Code snippets in languages *A* and *B* implementing the same test block are shown in **Figure 1** and **Figure 2**.

```
test_template_instance  t1  schema1
{
  binding1 = 0,
  binding2 = "operation1",
  binding3 = "-0.05, 0, 0.05"
}
```

Figure 1: Test Block in Language *A*

```
define-instance t1 template1 (
  (define param1 0 :choice "No")
  (define param2 operation1)
  (define param3 -0.05 0 0.05)
)
```

Figure 2: Test Block in Language *B*

Given a test specification file written in one vendor-specific language, our objective is to generate test specification files automatically in another vendor-specific language. To achieve this, the original test specification files need to be analysed to capture the vendor-independent data that will be translated.

3. Program Translators and Code Generation

Generative programming and program generators have been gaining attention among software engineers, especially for automating the creation of repetitive code. Cleaveland [1] advocates domain engineering, "a systematic process for understanding application domains and building tools for supporting software development in the application domain", for designing programming generators, or software that write other software. We share the viewpoint that this is an important step, and will use ontologies to aid us in this area, which is described in section 4.1.

Code generation is very commonly encountered in Web page creation, especially those using technologies such as Active Server Pages (ASP), PHP: Hypertext Preprocessor (PHP) or Java Servlet Pages (JSP) and involving Structured Query Language (SQL) scripts. In addition, many integrated development environments (IDE) and computer-aided software engineering (CASE) tools generate documentation, code snippets, user interfaces and even whole classes, based on user-defined values and preferences, Unified Modelling Language (UML) and entity-relationship (E-R) diagrams respectively. Van Wijngaarden and Visser's report [2] views code generation in terms of transformation, besides including an interesting and systematic comparison of several generation (or transformation) tools. One such tool is *Jostraca* [3] [4], which allows programmers to develop code templates using their choice of programming language, while mimicking JSP syntax.

In our translation engine, as are in the examples cited above, the code generation model has two important parts:

(i) the use of output text *boilerplates* (*how* to generate),
(ii) a declarative *model* of *what* to generate.

While the examples cited above generate code from a user-defined model, our problem requires us to *discover* the model from some existing software artefacts, i.e. the platform-specific test programs. The rest of this paper describes how our system framework is designed and implemented, based on the two ingredients above.

4. Design and Implementation of Translation Engine

Figure 3 depicts the overview of our translation engine model. The crux in our translation engine is a generic representation (*G*) for test programs, abstracted from the platform-specific languages. To perform a translation from source language *S* to target language *T*, the system needs to do the following:

(i) A *parser module* for language *S* first analyses the original test program to capture the information within. This results in a test specification file in *G*.
(ii) A *generation module* for language *T* then transforms the file in *G* to test specifications in language *T*.
(iii) The system pulls information from a *test domain ontology* to correctly translate test keywords on each platform in (i) and (ii).

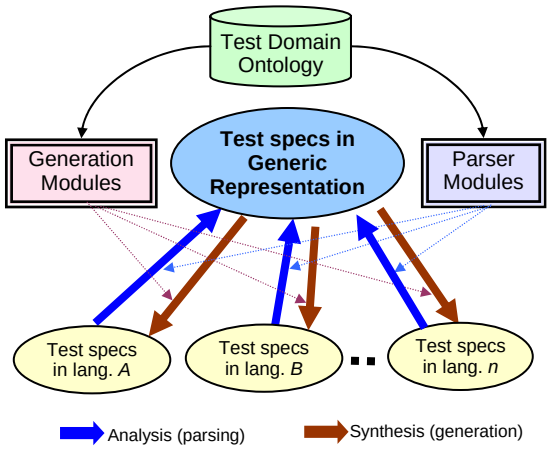

Figure 3: Overview of Architecture Model (2*n* modules)

Under this schema, which Waters [5] calls "abstraction-reimplementation", is similar to the interlingua approach for machine translation (i.e. automatic translation of natural language texts), the number of platform-specific modules to be developed when we have *n* platforms is 2*n*. If we had adopted a "direct transfer" approach (**Figure 4**), where a translation module is needed for each language pair, in each direction, *n*(*n-1*) modules would have been required instead.

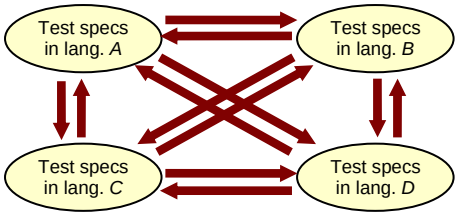

Figure 4: "Direct Transfer" Translation Approach (*n*(*n-1*) modules)

The interlingua approach did not work well for machine translation. Natural language is highly flexible and ambiguous, unlike formal languages. Consequently, attempts to represent "meaning" in natural language texts in a language-independent manner have not been successful [6]. Much language-specific linguistic details would be lost to generate correct outputs in the target language effectively. However, since our problem involves formal languages which are rigid and deterministic, this interlingua-like approach is expected to work well.

The following sections briefly describe the components as pictured in **Figure 5**, namely definition of the generic representation, the test domain ontology, the platform-specific parser and generation modules.

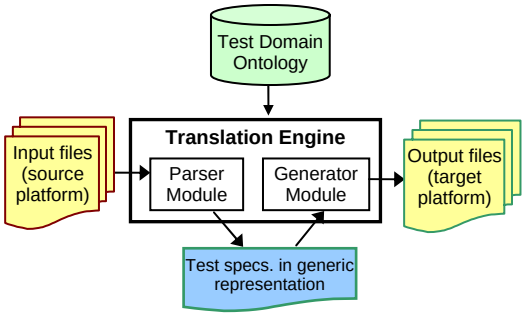

Figure 5: Translation Engine Components

4.1 Test Domain Ontology

The generic test specification language is the crux or "pivot" in our translation model. In order to effectively define this generic language, we modelled our knowledge, gleaned from studying the platform-specific languages, as an ontology.

An ontology is an "explicit formal specification of the terms in the domain and relations among them", by defining concepts, terms and vocabularies in a domain, as well as the relationships and restrictions among these concepts [7][8]. Ontologies are often used to share and distribute common understanding about a domain among people or software agents, as well as analysing knowledge in the domain through reasoning and inference [8]. They are particularly useful in areas where there is a need to standardise terms and vocabularies [9], such as (but not limited to) the medical and bioinformatics disciplines. (See [10] and [11] for a selection of existing ontologies for various domains, as well as the proceedings of past Protégé conferences [12] for their applications.)

Concepts in an ontology are organised in a taxonomy, similar to the familiar object-oriented paradigm. Ontology development and OO class design are nevertheless different, as noted by Noy and McGuiness in [8]: while OOP focuses on the *operational* aspects of a class, the emphasis of ontologies is on the *structural* properties of the class. Therefore, ontologies can complement object-oriented modeling during the requirements gathering, analysis and design stages, as recommended by the authors of [9], [13] and [14], and carried out by [15].

In our case (and that of [15]), the ontology aided us in achieving a better understanding of the problem (i.e. domain engineering in section 3), as well as providing a model for the generic language. It also became a knowledge base that facilitates our translation process. This confirms Jasper and Uschold's observation [9] that ontologies are effective in scenarios where "information authored in a single language is converted for multiple target platforms".

We constructed the test domain ontology as an OWL [16] file using *Protégé 2000* [17], an ontology editor tool. Besides capturing the key concepts (i.e. test schemas, test parameters and enumeration values) and restrictions about the test specifications, the ontology also captures mappings of test keywords on various platforms. Test engineers can also directly author generic test blocks, by modelling them as instances in the ontology, using the GUI editors offered by *Protégé*. The test blocks can then be exported to the generic representation scheme with a utility tool we implemented.

4.2 Generic Representation Scheme

We generically represent our platform-independent test specifications with XML (a choice also made by many authors and developers, including [1], [4], [18] and [19]), based on the key concepts identified in our test domain ontology. By choosing the XML format, we ensure that the generic test specifications are portable, is can be manipulated by a large range of widely available tools, and can be converted to various formats and outputs.

```
<test>
  <name>t1</name>
  <schema>sc1</schema>
  <binding name = "Bind1"
           enum="true" >0</binding>
  <binding name = "Bind2">
           operation1</binding>
  <binding name = "Bind3">
      <value>-0.05</value>
      <value>0</value>
      <value>0.05</value>
  </binding>
</test>
```

Figure 6: Test Block in Generic Representation

Figure 6 shows the same test block in **Figure 1** and **Figure 2**, but in the generic representation.

4.3 Parser Module

As indicated in **Figure 3**, a parser module is required for each platform-specific language to parse and analyse test programs in that language. To obtain the parser modules, context free grammar (CFG) specifications (with embedded parser actions) are written for each platform-specific language. This is possible as the platform-specific languages are all well-formed and formal, i.e. in the family of context free languages. We then used *JavaCC* [20] to process these CFG specifications and generate the parser classes.

During the parsing process (as pictured in **Figure 7**), each platform-specific parser module executes the following steps:

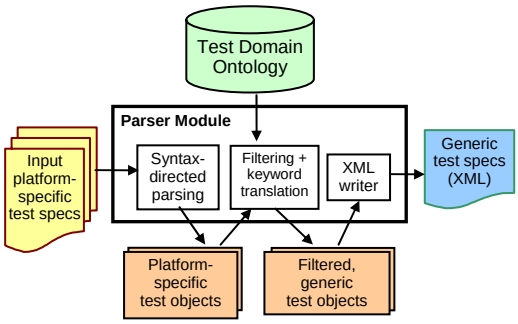

Figure 7: The Parsing Process

(i) **Syntax-directed parsing** of the input platform-specific test programs (**Figure 1** and **Figure 2**), following the CFG specifications written earlier. The parser creates in-memory test objects as a result of this parsing. All test keywords are *platform-specific* at this stage.

(ii) **Filtering and test keyword translation.** By referring to the test domain ontology, the parser module discards non-generic test objects, and translates the test keywords to their *platform-independent* forms.

(iii) **Writing to XML output.** Finally, the platform-independent test objects are written to an XML output file (using the generic representation scheme), which becomes the input to the generation process.

4.4 Generator Module

We take a templating approach in designing the generation module, using XSL stylesheets [21] to produce platform-specific test program syntax. XSLT transforms XML documents into various kinds of textual outputs. While it is commonly used to transform XML documents to HTML in web browsers or other XML documents, XSLT (together with XML) can also be deployed as a code generation tool, as demonstrated by Sarkar's article [19] and Mangano's book [22] on the subject.

We create an XSL stylesheet for each platform-specific language required. These XSL stylesheets come into play during the generation process, pictured in **Figure 8** and described below:

(i) **XML parsing.** The XML file containing the generic test specifications (**Figure 6**) is first parsed to obtain a document object model (DOM) tree with *platform-independent* test keywords.

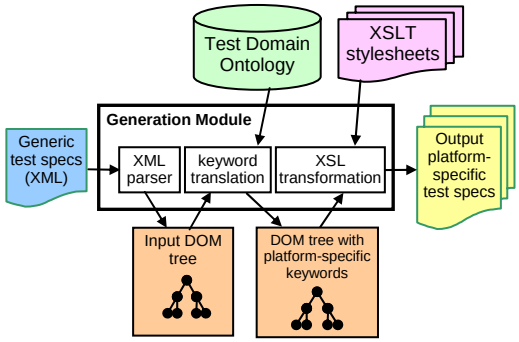

Figure 8: The Generation Process

(ii) **Test keyword translation.** The test keywords are replaced with their target *platform-specific* equivalents, by referring to the test domain ontology.

(iii) **XSL transformation.** The final step is to apply XSLT transformations on the DOM tree from step 0 using the XSL stylesheets, for the required target platform, thereby producing the final output (**Figure 1** and **Figure 2**).

5. Results

We have implemented the translation engine framework in the Java programming language, including the CFG specifications and XSL stylesheets for two platform-specific languages. Language *A* uses a C-like syntax similar to the code snippet shown in **Figure 1**, while language *B* uses a Lisp-like syntax, similar to that in **Figure 2**. To better illustrate the workings of the parser and generator module, **Figure 9** shows the transformations involved when translating the test specification block in **Figure 1** to that in **Figure 2**.

The input test specification file in language *A*, `test.A`, is first analysed by the parser classes

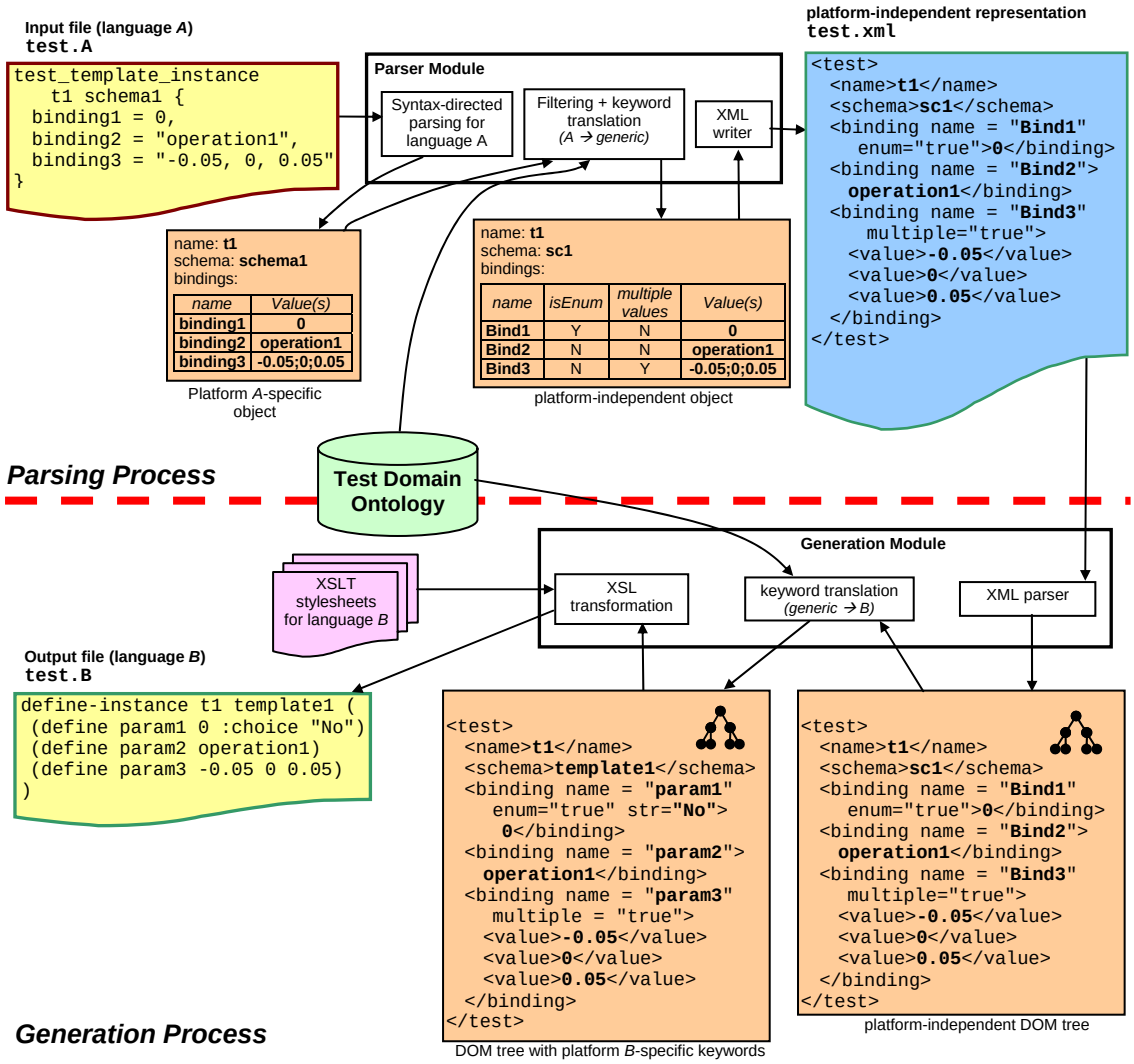

Figure 9: Example of Translating Test Specification from Language *A* to Language *B*

produced from the language *A* CFG, to obtain (in-memory) test objects representing the code blocks. After the platform *A*-specific keywords in these objects are translated to their platform-independent equivalents, the test objects are written to `test.xml` in the XML generic representation, concluding the parsing process.

During the generation process, `test.xml` is read into memory as a DOM tree. The generic test keywords are now translated to platform *B*-specific equivalents. Finally, XSLT stylesheets for platform *B* are applied to the DOM tree to produce the final output, `test.B`, i.e. the test specification file in language *B*.

Notice that the test domain ontology is used during the translation of keywords in all directions, and that it provides additional platform-specific information when necessary, e.g. the fact that platform *A*'s `binding1` actually takes on enumeration values, which also has a string value of `"No"` on platform *B*.

6. Future Work

While the prototype functions adequately, there is room for further development, especially in the following aspects:

(i) Adding support for dynamic platform-specific extensions.
(ii) Extending support to more types of code blocks: currently, our prototype engine only handles translation of two types of code blocks.
(iii) Extending support to more platform-specific languages.

We also expect any further development to involve the improvement and enhancement of the test domain ontology.

7. Conclusion

We have described and implemented a framework for translating test specification code between platform-specific languages. The

framework design is reusable for solving similar problems, i.e. scenarios involving translation of programming artefacts among different languages, formats or platforms.

Taking a leaf out of past work on machine translation, the use of a platform-independent language has allowed us to cut down on the number of modules to be developed. In developing the translation engine, we made use of CFG parser generators, as well as XML/XSLT technologies.

From our experiences, we also found ontologies to be useful tools for requirements gathering, analysis and design, as well as an alternative knowledge repository to databases.

8. Acknowledgements

The work described in this paper was sponsored by a research grant from Intel Technology Sdn Bhd. We are grateful to Mr Yong Suan Hueh of Intel for providing us with the sample test specifications and documentations, as well as helping us understand the specification files. We also thank the two anonymous reviewers for their comments on this paper.

9. References

[1] Cleaveland, J. C. Program Generators with XML and Java. Prentice-Hall PTR, New Jersey, 2001.

[2] van Wijngaarden, J. and Visser, E., "Program Transformation Mechanics: A Classification of Mechanisms for Transformation with a Survey of Existing Transformation Systems", *Technical Report UU-CS-2003-048*, Institute of Information and Computing Sciences, Utrecht University, Utrecht, the Netherlands, 2003.

[3] Jostraca. "The Jostraca Code Generator", http://www.jostraca.org/, 2005.

[4] Rodger, R. "Jostraca: a Template Engine for Generative Programming". *Position paper for the 16th European Conference on Object-Oriented Programming (ECOOP 2002) Workshop on Generative Programming*, Malága, Spain, 2002.

[5] Waters, R. C. "Program Translation via Abstraction and Reimplementation". *IEEE Transactions on Software Engineering*, 14(8), pp. 1207—1228, 1988.

[6] Hutchins, W. H. and Somers, H. L. An Introduction to Machine Translation. Academic Press Limited, London, 1992.

[7] Gruber, T. "A Translation Approach to Portable Ontology Specification", *Knowledge Acquisition*, 5(2), 1993, pp. 199-220.

[8] Noy, N. F. and McGuiness, D. L. "Ontology Development 101: A Guide to Creating Your First Ontology", *Technical Report SMI-2001-0880*, Stanford Medical Informatics, Stanford University, Stanford, 2001.

[9] Jasper, R. and Uschold, M. "A Framework for Understanding and Classifying Ontology Applications", *Proceedings of the 12th Banff Knowledge Acquisition for Knowledge Based Systems Workshop*, Banff, Alberta, Canada, 1999.

[10] Protege Ontologies Library. "ProtegeWiki: Protege Ontologies Library", http://protege.cim3.net/cgi-bin/wiki.pl?ProtegeOntologiesLibrary, 2005.

[11] OWL Ontologies. "Protégé OWL Plugin: Ontologies", http://protege.stanford.edu/plugins/owl/owl-library/index.html, 2005.

[12] Protégé Conferences. "Protégé Community: Protégé Conferences", http://protege.stanford.edu/community/conferences.html, 2005.

[13] Falbo, R. A., Guizzardi, G. and Duarte, K. C. "An Ontological Approach to Domain Engineering", *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE-2002)*, Ischia, Italy, 2002.

[14] Falbo, R. A., Guizzardi, G., Duarte, K. C. and Natali, A. C. C. "Developing Software for and with Reuse: An Ontological Approach", *Proceedings of ACIS International Conference on Computer Science, Software Engineering, Information Technology, e-Business, and Applications (CSITeA-02)*, Foz do Iguacu, Brazil, 2002.

[15] Trice, A. "Using Protégé in a Domain-Driven Software Product Development Process", *Proceedings of the 7th International Protégé Conference*, Bethesda, Maryland, 2004.

[16] OWL. "OWL Web Ontology Language Overview: W3C Recommendation", http://www.w3.org/TR/owl-features/, 2005.

[17] Protégé 2000. "The Protégé Ontology Editor and Knowledge Acquisition System", http://protege.stanford.edu/, 2005.

[18] Herrington, J. D. Code Generation in Action, Manning Publications, Connecticut, 2004.

[19] Sarkar, S. Model-Driven Programming using XSLT, *XML-Journal*, 3(8), 2002, pp. 42-51.

[20] JavaCC. "Java Compiler Compiler (JavaCC) - The Java Parser Generator", https://javacc.dev.java.net/, 2005.

[21] XSLT. "XSL Transformations (XSLT) Version 1.0. W3C Recommendation", http://www.w3.org/TR/xslt, 2005.

[22] Mangano, S. XSLT Cookbook, O'Reilly & Associates, California, 2002.