A Template-Based Approach to Modeling Variability

Soheila Bashardoust Tajali, Jean-Pierre Corriveau and Wei Shi
School of Computer Science, Carleton University
Colonel By Drive
Ottawa, CANADA
1-613-520-2600
{sbtajali, jeanpier, wei_shi}@scs.carleton.ca

ABSTRACT
Arnold and Corriveau have recently described ACL/VF, a non-state-based quality-driven approach to software specification that enables the requirements of stakeholders to be validated against the actual behavior of an implementation under test. Simultaneously, in recent years, the software product line (SPL) approach, initiated by Parnas back in the 1970s, has emerged as a promising way to improve software productivity and quality. The problem we address here can be summarized in one question: how can ACL/VF support product lines? The solution we propose adopts Cleaveland's template-based general approach to variability. We first explain how to go from the traditional feature diagram and feature grammar used in SPL to a) ACL domain contracts capturing commonalities between the requirements contracts of a domain and b) variability contracts capturing how features and their relationships (captured in a feature grammar) can affect these domain contracts. Domain and variability contracts are then captured in XML files and we rely on XSLT to specify how variability is to be resolved in order to generate a specific member contract.

Keywords— Variability, XSLT Template, Generators, Feature Diagrams

Contact author for SERP 2013 paper: J-Pierre Corriveau

1. ON GENERATIVE APPROACHES TO VARIABILITY

In recent years, the software product line (SPL) approach [7, 8, 9] initiated by Parnas back in the 1970s [17] has emerged as a promising way to improving software productivity and quality. A product line, which corresponds to a *domain*, arises from situations when we need to develop multiple similar products. A *commonality* is a property held across this domain, whereas a *variant* (or *variability*) is a property specific to certain *members* of this domain. Most importantly, Zhang and Jarzabek [20] remark that "the explosion of possible variant combinations and complicated variant relationships make the manual, ad hoc accommodation and configuration of variants difficult." Thus, it is generally agreed that a variability mechanism that supports automated customization and assembly of product line assets is required. Consequently, a significant amount of work has focused on the creation of *generators* to automate going from a model of variability to a specific member of a family of products. Let us elaborate.

With respect to terminology, we will adopt the one of Czarnecki and Eisenecker [9]: In System Family Engineering (or equivalently, Software Product Lines), *members* of a *domain* share a set of common *features*, as well as possibly possessing their specific ones. Commonalities, we repeat, refer to the characteristics that are common to all family members, while variabilities distinguish the members of a family from each other and need to be explicitly modeled and separated from the common parts. Conceptually, a feature is a *variation point* in a space of requirements (the domain) and has several *variants* (also called feature values) associated with it. The two main processes of SPL engineering are a) domain engineering (for analyzing the commonality and variability between members) and application engineering (for generating individual members of the domain).

Domain engineering rests on the creation of a domain model via feature modeling [9]. Conceptually, application engineering then consists in defining a specific *configuration* of feature values and generating from the domain model and from this configuration the corresponding member (of the domain). Thus, SPL engineering is a model-driven activity involving both the modeling of commonalities and variabilities of a domain, and the generation of a member of this domain from this model. Many languages and approaches have been proposed for modeling variability (see [4] and [18] for recent in-depth reviews). As for approaches to generation, they can be separated into two categories, each including many proposals:

- *transformational* methods, which define explicit mappings between semantic elements of a source model and those of a target model.

- *generative* approaches, which build a target model from what amounts to a parameterized source model and a configuration list (that supplies specific values for these parameters).

It has been argued that generative approaches correspond to a more powerful semantic approach to the production of a target model than transformations [7, 9]. This paper focuses on the creation of a particular generator. We first introduce the specific problem we address, then overview the solution we propose for it.

2. PREMISES

A quality-driven approach to software development and testing demands that, ultimately, the requirements of stakeholders be validated against the actual behavior of an Implementation Under Test (hereafter IUT). That is, there needs to be a systematic (ideally objective and automated) approach to the validation of the requirements of the stakeholder against the actual behavior of an IUT [3]. Unfortunately, such systematic approach to validation remains problematic [5, 11] and, in practice, testers mostly carry out only extensive unit testing [6, 16].

In order to validate the requirements of a stakeholder against the actual behavior of an IUT, it is necessary to have a specification language from which tests can be generated and executed 'against' an actual IUT (as opposed to a model of the latter). Arnold and Corriveau have described at length elsewhere [1] such an approach and its corresponding tool, the Validation Framework (hereafter VF [2]).

The VF operates on three input elements. The first element is the Testable Requirements Model (hereafter TRM). This model is expressed in ACL, a high-level general-purpose requirements contract language. We use here the word 'contract' because a TRM is formed of a set of contracts, as illustrated shortly. ACL is closely tied to requirements by defining syntax/semantics for the representation of scenarios, and design-by-contract constructs [15] such as pre and post-conditions, and invariants (rooted in [12, 13]).

The second input element is the candidate IUT against which the TRM will be executed. This IUT is a .NET executable (for which no source code is required).

Bindings represent the third and final input element required by the VF. Before a TRM can be executed, the types, responsibilities, and observability requirements of the TRM (see example below) must be bound to concrete implementation artifacts located within the IUT. A structural representation of the IUT is first obtained automatically. The binding tool, which is part of the VF, uses this structural representation to map elements from the TRM to types and procedures defined within the candidate IUT. In particular, this binding tool is able to automatically infer most of the bindings required between a TRM and an IUT [1, 2, 3]. Such bindings are crucial for three reasons. First, they allow the TRM to be independent of implementation details, as specific type and procedure names used with the candidate IUT do not have to exist within the TRM. Second, because each IUT has its own bindings to a TRM, several candidate IUTs can be tested against a single TRM. Finally, bindings provide explicit traceability between a TRM and IUT.

Once the TRM has been specified and bound to a candidate IUT, the TRM is compiled. Upon a successful compilation, all elements of the TRM have been bound to IUT artifacts. The result of such a compilation is a single file that contains all information required to execute the TRM against a candidate IUT. The validation of a TRM begins with a structural analysis of the candidate IUT, and with the execution of any static checks (e.g., a type inherits from another). Following execution of the static checks, the VF starts and monitors the execution of the IUT. The VF is able to track and record the execution paths generated by the IUT, as well as execute any dynamic checks, and gather user-specified metrics specified in the TRM. The execution paths are used to determine if each scenario execution matches the grammar of responsibilities corresponding to it within the TRM.

The key point of this overview is that once a TRM is automatically bound to an IUT, all checks are automatically instrumented in the IUT whose execution is also controlled by the VF. This enables verifying that actual sequences of procedures occurring during an execution of an IUT 'obey' the grammar of valid sequences defined in ACL scenarios. Most importantly, no glue code (that is, code to bridge between test specifications and actual tests coded to use the IUT) is required.

The problem we address in this paper can be summarized in one question: how can ACL/VF support domain engineering and application engineering? In the specific context of ACL/VF, this question can be broken down into two more immediate ones:

1) how can ACL (i.e., the requirements modeling language) be 'augmented' to support some modeling of variability?

2) how can such augmented ACL models be used, together with some specification of a configuration of feature values, to generate a domain *member* contract, that is, the set of contracts associated with a specific member of a domain?

An answer to these questions requires that the reader first get a basic understanding of the syntax and semantics of ACL. To this end, we give below a short self-explanatory example:

```
Namespace My.Examples

{

/*Each ACL contract is bound to one or more
types of the IUT. An ACL contract may define
variables, which will be stored and updated
by the VF. */

Contract ContainerBase<Type T>

/* The variable size tracks the number of
elements in a container according to the ACL
model. It is NOT associated or dependent on
any similar variable(s) in the IUT. */

{  Scalar Integer size;
/*An observability is a query-method that is
used to request state information from the
IUT. That is, they are read-only methods
that acquire and return a value stored by
the  IUT. An observability thus defines some
data that the IUT MUST be able to supply to
the VF for the VF to properly monitor the
IUT.*/
  Observability Boolean    IsFull();
  Observability Boolean    IsEmpty();
  Observability T ItemAt(Integer index);
  Observability Integer    Size();
Responsibility new()
{ size = 0;   Post(IsEmpty() == true)
       }
Responsibility finalize()
{ Pre(IsEmpty() == true);  }
Invariant SizeCheck
{ Check(context.size >= 0);
   Check(context.size == Size())        }
/* The next responsibility defines pre- and
post- conditions for addition. It is not to
be bound but rather to be extended by actual
responsibilities.   The   keyword   'Execute'
indicates where execution occurs. */
Responsibility Add(T aItem)
{ Pre(aItem not= null);
   Pre(IsFull() == false);  Execute();
   size = size + 1;
  Post(HasItem(aItem));           }
/*This   responsibility   extends   Add.   It
therefore   reuses   its   pre-   and   post-
conditions of Add. */
Responsibility  InsertAt(Integer   index,  T
aItem)
       extends Add(aItem)
{      Pre(index >= 0);    Execute();
       Post(ItemAt(index) == aItem);       }
/*   other   responsibilties   for   adding,
removing, searching, etc. are ommitted here.
/*
Scenario AddAndRemove
{ once Scalar T x;
   Trigger(Add(x) | Insert(dontcare, x)),
   Terminate((x      ==      Remove())      |
(RemoveElement(x)));  }
}
/* The Export section defines the types used
in   this   contract,   as   well   as   their
constraints. */
Exports
{ Type tItem conforms Item
       { not context;   not derived context;
       }      } }
}  //end of contract ContainerBase
```

This single TRM has been applied to several simple data structures (e.g., different kinds of arrays and linked lists) implemented in C# and C++/CLI, with and without coding errors, in order to verify that ACL/VF indeed detects responsibility and scenario violations. This ability to bind the same TRM against several distinct IUTs may mislead some readers to believe ACL/VF already handles some form of variability. In fact, it does not: all the IUTs that can bind to this TRM can do so specifically because there is **no** variablity in the TRM they must conform with. In other words, regardless of their differences at the level of code, all the IUTs that can bind to this TRM can do so because they have been adapted to support the observabilities required by this TRM.

So the question remains: how can ACL/VF be augmented to support variability? As previously mentioned, many languages and approaches have been proposed for modeling variability. But few are relevant to this work due to a fundamental restriction we are faced with: neither the ACL nor the VF can be modified. That is, given the ACL/VF is an experimental tool of over 250,000 lines of code, which is still undergoing testing, we decided to support variability in ACL contracts *without* altering the syntax or semantics of ACL, or the working of the ACL compiler, or the modus operandi of the VF.

To achieve this goal, we adopt Cleaveland's [7] template-based general approach to variability, captured in Figure 1.

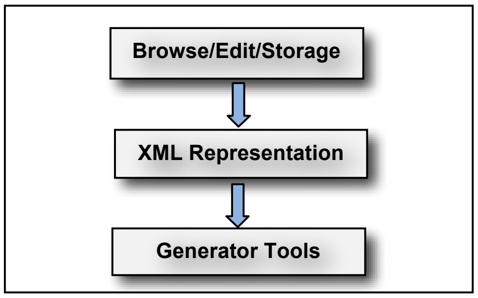

Figure 1: Cleaveland's Generative Approach

In a nutshell, following this approach, modeling variability is a task that ultimately must produce an XML representation of variabilities and commonalities of the domain. It is this representation that is used to generate a specific member of the domain. The advantage of choosing XML is that it makes available the much wider world of XML technologies and tools. In particular, XSLT is a standard transformation language for transforming between XML languages or to other text-based languages. That is, XSLT is readily usable for creating a generator. Czarnecki [10] explains (in the specific context of code generation):

"In a template-based generative approach (a) an arbitrary text file such as a source program file in any programming language or a documentation file is instrumented with code selection and iterative code expansion constructs. The instrumented file called template needs a template processor. A template processor takes a template file and a set of configuration parameters as inputs and generates a concrete instance of that template as output".

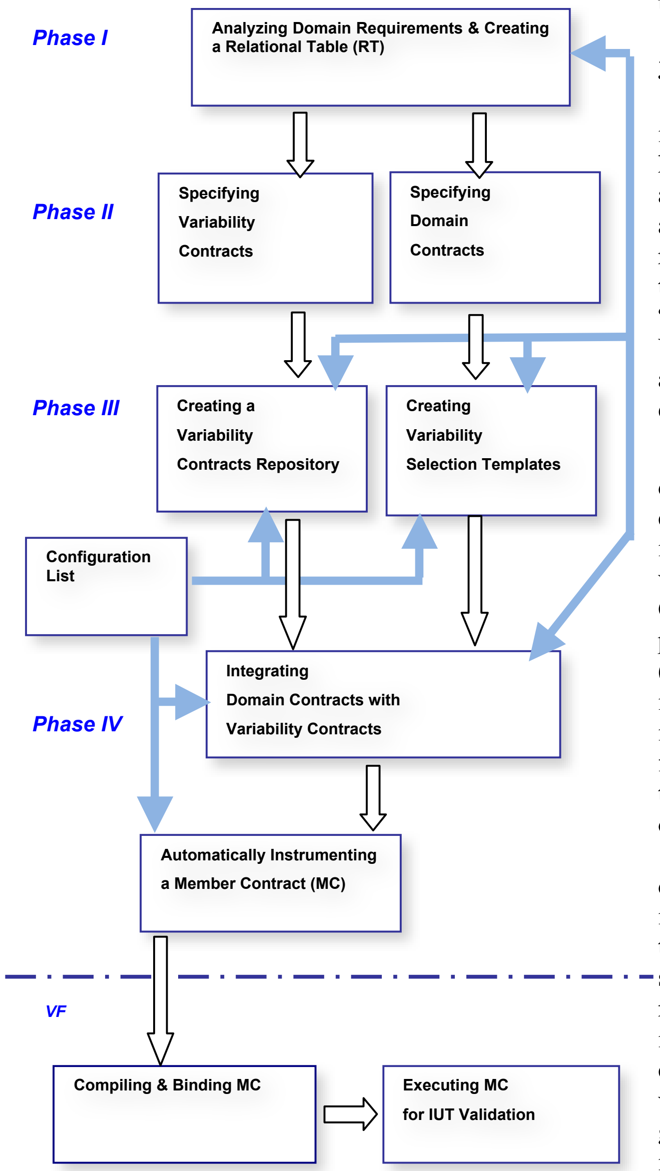
Phase I
Analyzing Domain Requirements & Creating a Relational Table (RT)
Phase II
Specifying Variability Contracts
Specifying Domain Contracts
Phase III
Creating a Variability Contracts Repository
Creating Variability Selection Templates
Configuration List
Phase IV
Integrating Domain Contracts with Variability Contracts
Automatically Instrumenting a Member Contract (MC)
VF
Compiling & Binding MC
Executing MC for IUT Validation

Figure 2: Overview of our Generative Approach

In contrast, a programming language based generative approach, such as Czarnecki and Eisenecker's [9] C++ metaprogramming, typically uses advanced programming techniques (such as partial template specialization in C++) that are not only hard to master and problematic to debug (leading to complex generators), but also do not offer as much semantic flexibility as an XML based approach. For these reasons, the generative approach we propose (see Figure 2) for tackling variability in ACL adopts a template-based approach rooted in XML. In the rest of this paper, we overview the different steps identified in Figure 2 (in which the white arrows show the evolution from one artifact to another, and the blue arrows on the right of the figure indicate where user input is required, in contrast to those on the left that show where the configuration list is used).

3. DETAILS

In Phase I of our approach, the domain engineer first identifies commonality and variability in the domain at hand. This analysis of the domain leads to the production of a *feature model* [9]. The syntax and semantics of this model are that of FODA [14] and similar notations [see 9 for a review]. A feature diagram captures variation points and their variants. Here, for example, variation point VP1, "Length Type", has two variants VP1-1 "Variable" and VP1-2 "Fixed". It captures the fact that containers can have a variable length or fixed length, where length is number of elements.[1]

While a feature diagram is a good starting point for domain analysis, it is crucial to understand that the complexity of a generative approach lies first and foremost in its handling of interactions between features and feature values, which are captured in a *feature grammar* [9]. Consequently, it is unfortunately often the case that the processing of such a feature grammar is entirely manual (e.g., [18]). In contrast, in our solution, the feature interactions identified by the domain engineer are captured in a table whose use is automated. We call such a table a feature grammar table, or equivalently a feature relational table (RT). Table 1 (next page) presents a few of the rows of the large RT developed for one of our case studies. Rows 1 and 11 of the complete table [4] are examples of how to define the exact relationship between a variation point and its variants. For example, row 1 below states that VP0 has two mutually exclusive variants VP0-1 and VP0-2. It also states that if the configuration list (defining a specific member of the domain through a specific set of variants) includes one of the variants of VP-0, there are no conditions to verify and this variant can be taken as the value of VP0 (for further verification of the feature grammar, as well as subsequent generation). Rows 13 to 21 in the complete table [4] deal with feature interactions proper. Row 13 below, for example, states that if VP31 is assigned any one of its valid variants in the configuration list, then VP30 must also appear in this configuration and must be set to variant LC-Type. VP30 is an optional feature

[1] We also refer to VP0 (whether a container is key-based or not), VP4 (whether a container is circular, VP4-1 or not, VP4-2), and VP5 (whether a container allows 2-way, VP5-1, or 1-way traversal, VP5-2).

capturing whether or not the container keeps track, via a counter, of the number of its elements. VP31 merely captures the type of this counter.

Rows 18 and 19 illustrate how multiple valid combinations of variants are handled: one action is associated with a circular container supporting two-way traversal (row 18) and another action is used for a circular container supporting only one-way traversal (row 19). The key point to be grasped is that all such valid combinations of features and variants must be explicitly captured in this feature grammar (according to specific syntax and semantics defined in [4] and similar to FODA).

In Phase II of our approach, first, the commonalities of the domain at hand are captured in ACL contracts forming the *domain contracts*. These contracts are merely ACL contracts augmented in Phase III with plugIn labels (see Figure 3) marking where variability (and thus the generator) may modify such contracts. In Figure 3, we are simply indicating that *IsFull* may be affected by the variant chosen for VP1. (Several variation points may affect the same element of an ACL contract.)

The feature diagram and relationship table of phase I also lead, in Phase II, to *variability contracts* (via algorithms we have developed for this specific purpose [4]). The idea is to capture how each variant of a variation point affects the domain contract. Without going into (syntactic and semantic) details (given in [4]), a portion of such a variability contract is shown in Figure 4. Intuitively, if length is variable (VP1-1), then the *IsFull* observability necessarily returns false, otherwise (VP1-2), it returns whether the actual size of the container has reached the maximum size.

```
Observability Boolean IsFull()
       {      //plugin_VP1();    }
```

Figure 3: Variability in a Domain Contract

```
 Variation VP1 <Length-Type> [1..1] outof 2
 {case "Variable":
   plug-in: VP1-1 //Container has variable length
   Refine-a: Observability
   Boolean IsFull(){ value = false; }
  case "Fixed":
    plug-in: VP1-2 //Container has fixed length
    Refine-a: Observability
    Boolean IsFull() { value = (size() == max_size()); }
}
```

Figure 4: A Variability Contract

It must be emphasized that there is a direct correspondence between the features and variants identified in the feature diagram and relational table of Phase I, and these variability contracts (as well as the plugIn labels of domain contracts). Conceptually, variability contracts define the actions to be performed (by the generator) on the template (i.e., the domain contracts in our work) when a particular feature value is present in a configuration list input to generate a specific member of the domain. A template processor (i.e., our chosen kind of generator) can carry out such actions only if the domain contracts define where these actions are to take place (and thus the need for the plugIn labels we introduced). That is, as in other template-based generative approaches, the artifact capturing the domain model must be *instrumented* (to use Czarnecki's terminology) to indicate where in it template manipulations can occur. Then, both the domain and the variability contracts must be transformed into their XML equivalents in order to be made usable by the template processor. This is what Phase III of our approach tackles:.

Table 1: A portion of a Relation Table for Sequential Containers

No	Related VP(s) Var(s)	Related VP & Var Types	Related VP & Var Names	Relation: Rule# Constraints & Actions	Relation: <depends>, <requires>,<excludes> Constraints and Actions in Contracts
1	VP0 VP0-1 VP0-2	Var. point Variant Variant	“Is-key-based” ”True” “False”	Rule 1, 2: VP0 ←X VP0-1 VP0 ←X VP0-2	VP0 <depends>VP0-1, VP0-2 Cond = - Action = variant
11	VP31 VP31-1 VP31-2 VP32-3	Var. point Variant Variant Variant	“LC-Type” “int” “short” “long”	Rule 1, 2: VP31 ←X VP31-1 VP31 ←X VP31-2 VP31 ←X VP31-3	VP31<depends>VP31-1, VP31-2, VP31-3 Cond = - Action = variant
13	VP31 VP31-1 VP31-2	Var. point Variant Variant	_	Rule 7: If(VP31 !=null && VP31 ==VP31-1 ||	VP31 <requires>VP30? = VP31 Cond1 = (VP30? == LC-Type)

	VP31-3 VP30	Variant Var. point		VP31 ==VP31-2 || VP31 ==VP31-3) IMPLIES VP30?==LC-Type	Action = variant1
18	VP4 VP4-1 VP5 VP5-1	Var. point Variant Var. point Variant	_	Rule 5: If(VP4==VP4-1) IMPLIES VP5 == VP5-1	VP4-1<requires>VP5-1 Cond1="VP5==Twoway" Action= variant1
19	VP4 VP4-1 VP5 VP5-2	Var. point Variant Var. point Variant	_	Rule 5: If(VP4==VP4-1) IMPLIES VP5 == VP5-2	VP4-1<requires>VP5-2 Cond2="VP5==Oneway" Action= variant2

a) The variability contracts of Phase II are transformed (again according to specific algorithms) into an XML-ready repository of these contracts.

b) Everywhere in the domain contracts where a plugIn label is used, a *variability selection template* is inserted. Given our specific approach to template processing, in our work these variability selection templates take the form of XSLT style sheets as explained at length in [4].

Finally, in order for these selection templates to be able to use the information of the variability contract repository, we still need to augment the domain contracts with XSLT code to bridge to the variability contract repository. At the end of phase III, both the variability contracts and the domain model have been transformed into XML-based artifacts that serve as input to the template-based generator, which also requires a configuration list specifying the exact list of feature values (i.e., the configuration) to be used to generate a specific member of the domain. Phase IV of our approach deals with the generation of such a member contract, that is, of an ACL contract corresponding to the specific input configuration list at hand. In a nutshell, the configuration list, as well as the variability contracts and selection templates (compiled using both an XML and an XSLT compiler) are integrated. The resulting ACL contract can then serve as input to ACL/VF so that it can be compiled and tested.

4. CONCLUSION

The main contribution of this work is a domain-independent generative process we propose for obtaining ACL member contracts from ACL-based domain and variability contracts. It is worth repeating that this process is comprehensive inasmuch as it addresses how the two traditional artifacts of domain engineering, namely a feature diagram and a feature grammar, can be evolved into domain and variability contracts whose XML equivalents serve as inputs (along with a configuration list) to the proposed generative process, which generates a member's contract (that can be compiled and run in ACL/VF).

ACL/VF is still an experimental model-based testing tool at this point in time, with advantages and drawbacks that its creators have discussed elsewhere [3]. We chose it to illustrate the power and generality of the template-based approach to generation advocated by Cleaveland [7]) for two main reasons:

1) It's a textual requirements language and ACL/VF already produced an XML equivalent of the ACL contracts specified by a user [1]. (Dealing with a visual language is somewhat more complex.)

2) The semantics of ACL are sufficiently comprehensive to tackle domain modeling and yet, most importantly, almost all of ACL's semantic elements (e.g., responsibilities, scenarios, observabilities, etc.) are relevant to variability.

Furthermore, we stress that, in contrast to many existing generative approaches, we have not only defined the artifacts relevant to the generative process but, most importantly, we also specified elsewhere [4] detailed algorithms to go from the more abstracts artifacts to those directly used by the generator. In fact, these algorithms inherently define traceability between the different artifacts of Figure 2. In turn, such traceability is essential to support an iterative approach to domain and application engineering. We also emphasize that, in contrast to many existing approaches (e.g., [18]), these algorithms do not assume that the user only inputs valid configurations. Such an assumption is a gross oversimplification: in our opinion, 'enforcing' that a configuration does respect the rules of a feature grammar must be automated, as is the case in our solution. Similarly, our work does not depend on any notion of the 'semantic correctness' of a feature diagram, feature grammar or domain contract supplied by a user. Such notion appears quite problematic [1].

Finally, the validation of our solution for the generation of an ACL member contract from domain contracts, variability contracts and a configuration specific to that member rests two extensive case studies. Both case studies [4] pertain to containers, reflecting the fact that the use of off-the-shelf component (COTS) libraries is pervasive in current software development processes. The first case study focuses specifically on sequential containers (such as arrays, lists, stacks and queues). This choice was straightforward given existing work [9, 19] on feature modeling across a large set of such container libraries (including those found in the Standard Template Library of C++). In other words, we wanted to avoid the all-too-frequent 'toy' example in favor of a realistic example based on public domain libraries. For our second case study, our focus was specifically on exercising more of the mechanisms we had developed for variability contracts. To do so we decided to tackle another facet of the STL, namely associated containers (such as dictionaries, multisets, etc.). The point we want to emphasize is that having an actual code base to take inspiration from for domain modeling eliminated the risk of creating an artificial domain conveniently scoped to work with our proposal. But this choice also meant tackling the modeling of some of the complexities of actual industrial code.

We have now turned our attention to the modeling of variability in design patterns, a radically different domain, in order to demonstrate that our proposal is not domain dependent.

5. REFERENCES

[1] Arnold, D.: “Supporting Generative Contracts in .Net”, Doctoral dissertation, School of Computer Science, Carleton University, April 2009.

[2] Arnold, D.: “Validation Framework and Another Contract Language”, http://vf.davearnold.ca/, last accessed in October 2012.

[3] Arnold, D., Corriveau, J.-P., Shi, W.: “Modeling and Validating Requirements using Executable Contracts and Scenarios", SERA 2010: 311-320.

[4] Bashardout-Tajali, S., Generative Contracts, Doctoral Dissertation, School of Computer Science, Carleton University, December 2012.

[5] Bertolino, A.: “Software Testing Research: Achievements, Challenges and Dreams”, In IEEE – Future of Software Engineering (FOSE ’07), Minneapolis, pp. 85-103, May 2007

[6] Binder, R.: *Testing Object-Oriented Systems*, Addison-Wesley Professional, Reading, MA, 2000.

[7] Cleaveland, C.: “Program Generators with XML and Java”, Prentice-Hall, Upper Saddle River, NJ, ISBN-10: 0130258784, Jan. 2001.

[8] Coplien, J., Hoffman, D., Weiss, D.: “Commonality and Variability in Software Engineering”, Bell Labs, IEEE Software, pp.37-45, Dec. 1998.

[9] Czarnecki, K., Eisenecker, U.W.: “Generative Programming: Methods, Tools, and Applications”, Addison-wesley, Boston, MA, 2000.

[10] Czarnecki, K., Bednasch, T., Unger, P., Eisenecker, U.: “Generative Programming for Embedded Software: An Industrial Experience Report”, In Don Batory et al., editors, Generative Programming and Component Engineering: ACM SIGPLAN/SIGSOFT Conference, GPCE 2002, LNCS 2487, Pittsburgh, PA, USA, pp. 156-172, October 2002.

[11] Grieskamp, W.: “Multi-Paradigmatic Model-Based Testing”, Technical Report #MSR-TR-2006-111, Microsoft Research, August 2006.

[12] Helm, R., Holland, I. M., Gangopadhyay, D.: “Contracts: Specifying Behavioural Compositions in Object-Oriented Systems”, In Proceedings of the ACM European conference on object-oriented programming systems, languages, and applications, Conference (OOPSLA'90), pp. 169-180, October 1990.

[13] Holland, I. M.: “Specifying reusable components using Contracts”, In Proceedings of the 6th European Conference on Object-oriented Programming (ECOOP ’92), pp. 287-308, 1992, LNCS/615.

[14] Kang K., Cohen S., Hess J., Novak W., Peterson A.: “Feature-Oriented Domain Analysis (FODA) Feasibility Study”, Technical report, CMU/SEI-90-TR-021, November 1990.

[15] Meyer, B.: “Design by Contract”, In *IEEE Computer*, vol. 25, no. 10, pp. 40-51, IEEE Press, New York, October 1992.

[16] Meszaros, G.: “xUnit Test Patterns: Refactoring Test Code”, Addison-Wesley Professional, ISBN-10: 0131495054, 2007.

[17] Parnas, D.L.: “On the Design and Development of Program Families”, IEEE Trans. Software Engineering, vol. 2, no, 1, pp. 1-9, March 1976.

[18] Tawhid, R.: “Integrating Performance Analysis in Model Driven Software Product Line Engineering”, Doctoral dissertation, School of Computer Science, Carleton University, May 2012.

[19] Tian, B., Corriveau, J.-P.: “On Facilitating the Reuse of C++ Graph Libraries”, In Proceedings of the IASTED International Conference on Software Engineering, part of the 23rd Multi-Conference on Applied Informatics, Innsbruck, Austria, pp. 7-12, February 2005.

[20] Zhang, H., Jarzabek, S.: “XVCL: A Mechanism for Handling Variants in Software Product Lines”, Special issue on Software Variability Management of Science of Computer Programming, Vol. 53, No. 3, pp. 381–407, December 2004.