Generative Contracts

by

Soheila Bashardoust Tajali

B.C.S., M.C.S.

A Thesis submitted to the

Faculty of Graduate and Postdoctoral Affairs

in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

in

Computer Science

Ottawa-Carleton Institute for Computer Science

School of Computer Science

Carleton University

Ottawa, Ontario

November, 2012

Your file Votre référence
ISBN: 978-0-494-94208-6

Our file Notre référence
ISBN: 978-0-494-94208-6

Abstract

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. To do so, 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). Corriveau and Arnold have recently described at length elsewhere such an approach (in the form of a contract-based specification language called ACL (Another Contrcat Language)) and its corresponding tool, the Validation Framework (hereafter VF).

Simultaneously, in recent years, the software product line approach, initiated by Parnas back in the 1970s, has emerged as a promising way to improve software productivity and quality. A product line (also called a 'product family' or 'system family') arises from situations when we need to develop multiple similar products (e.g., for a library, or for different clients, or from a single system over years of evolution).

The problem we address in this dissertation 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?

The solution we propose adopts Cleaveland's template-based general approach to variability. We explain at length and demonstrate with two case studies how to go from the traditional feature diagram and feature grammar used in System Family Engineering and Software Product Lines to a) domain contracts capturing commonalities between the requirements contracts of a domain and b) variability contracts capturing how features and their relationships (captured in the feature grammar) can affect these domain contracts. Domain and variability contracts are ultimately captured in XML files relying on XSLT to specify how variability is to be resolved to generate a specific member contract. This generative process uses off-the-shelf XML technologies to generate ACL contracts that indeed can be input, compiled and run in ACL/VF. Our solution is domain independent and we believe that being template-based it is not specific to ACL and can be generalized (though this claim is left as future work).

To my beloved parents, Soheil and Aghdas,
for their endless love and support

In memory of
my beloved father, Soheil and
my dear aunt, Anis

Acknowledgments

I express my deep gratitude to my advisor Prof. J.-P. Corriveau, for his guidance, support, and encouragement during my years at Carleton. Over the last six years, he challenged me to be better, supported me to move forward, and encouraged me to strive for excellence. His valuable advice and insights guided me throughout my Ph.D. study and will continue to guide me through my future career. Dr. Corriveau, I really thank you for everything.

I would like to thank Dr. Abhari, Dr. Ajila, Dr. Some, and Dr. Shi for being in my defense committee and for their invaluable suggestions and comments. I also thank all my teachers, professors, and paper reviewers during the years of my studies, to whom I will always be in debt.

I would like to express my appreciation to all office administrators at the School of Computer Science, who were available anytime for help during my program. My special thanks to Linda, Anna, Claire, Sharmi, and Edina.

My special thanks to my friend and former colleague at our favorite IS Lab Voja Radonjic, for his helpful discussions, critical reviews of papers and technical reports, and companionship through the years. He has provided a refuge for me away from the pressures of work and research during all those hard years. I would like to thank all my friends and lab-mates for making our lives more colorful and enjoyable at IS Lab.

Finally, I would like to thank my parents for their endless love, generosity, and support. I dedicate this dissertation to them. I also thank my brothers Masoud and Siamak, and my sister Maryam, for their patience, kindness, care and valuable encouragement through the hard times. I am greatly indebted to my family for their unconditional love and support without which my studies at Carleton might have never happened.

Table of Contents

List of Tables

List of Figures

List of Acronyms

ACL	Another Contract Language
ACL-V	ACL for Variability
ATL	Atlas Transformation Language
BNF	Backus-Naur Form
CER	Contract Evaluation Report
CIL	Contract Intermediate Language
CVM	Consolidated Variability Model
DLL	Dynamic Link Library
DSL	Domain Specific Language
EBNF	Extended Backus-Naur Form
EMF	Eclipse Modeling Framework
FODA	Feature Oriented Domain Analysis
GRL	Goal-oriented Requirement Language
IDE	Integrated Development Environment
IUT	Implementation Under Test
MC	Member Contract
MDA	Model Driven Architecture
MDD	Model Driven Development
MDE	Model Driven Engineering
MDG	Model Driven Generative
MDT	Model Driven Transformation
MOF	Meta-Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
OVM	Orthogonal Variability Model
PL	Product Line
QVT	Query/View/Transformations
RE	Requirement Engineering
SFE	System Family Engineering
SPL	Software Product Line
STL	Standard Template Library
TLV	Traceability Links for Vriability
TRM	Testable Requirements Model
UCM	Use Case Map
UML	Unified Modeling Language
URN	User Requirement Notation
VF	Validation Framework
VML	Variability Management Language
XML	Extensible Mark-up Language
XSLT	XSL Style-sheet

1 Introduction

Coplien et al. [COP98] write: "Increasingly, software engineers spend their time creating software families consisting of similar systems with many variations [...] Scope, Commonality, and Variability (SCV) analysis gives software engineers a systematic way of thinking about and identifying the product family they are creating [...]. A commonality is an assumption held uniformly across a given set of objects (S). Frequently, such assumptions are attributes with the same values for all elements of S. Conversely, a variability is an assumption true of only some elements of S, or an attribute with different values for at least two elements of S."

Indeed, in recent years, the software product line (SPL) approach [CLEA01], [CZA00], initiated by Parnas back in the 1970s [PAR76], has emerged as a promising way to improving software productivity and quality. A product line (also called a 'product family' or 'system family') arises from situations when we need to develop multiple similar products (e.g., for a library, or for different clients, or from a single system over years of evolution). Zhang and Jarzabek [ZHA04] add: "Members of a product line share many common requirements and characteristics. They may perform similar tasks, exhibit similar behavior or use similar technologies. While having much in common, members of a product line also differ in certain requirements, design decisions and implementation details [...] We identify both commonality and variability in a domain, and build generic and adaptable assets such as the domain model, product line architecture and generic components. In the development of each specific product, we reuse the product line assets instead of working from scratch [...] The explosion of possible variant combinations and complicated variant relationships make the manual, ad hoc accommodation and configuration of variants difficult [...] An effective way to deal with the problem of handling variants is to design a variability mechanism that supports automated customization and assembly of product line assets." Consequently, a significant amount of work (see chapter 2) 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 [CZA00]: In System Family Engineering (or equivalently, Software Product Lines) (hereafter SFE/SPL), members of a *domain* share a set of common *features*, as well as possibly possessing their specific ones. *Commonalities* refers to the characteristics that are common to all family members, while *variabilities* distinguish the members of a family from each other and needs 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 SFE/SPL engineering are:

- *domain engineering* for analyzing the commonality and variability between members,
- *application engineering* for *generating* individual members of the domain.

Domain engineering rests on the creation of a domain model via feature modeling. Conceptually, application engineering then consists in specifying a specific *configuration* of feature values and generating from the domain model and from this configuration the corresponding member (of the domain). Thus, SFE/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 chapter 2). As for approaches to generation, they can be separated into two categories, as discussed at length in chapter 2:

- *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). Thus, in practice, generative approaches

correspond to a much more powerful semantic approach to the production of a target model than what mappings offer.

Both categories include many proposals and are reviewed at length in the next chapter. In the rest of this chapter, we first introduce the specific problem we address, then present a brief overview of the solution we propose for it. We follow with a summary of the contributions of our work and a plan for the rest of this dissertation.

1.1 The Problem

1.1.1 ACL and VF

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 [ARN10]. Unfortunately, most often, there is no such systematic approach to validation [BERT07], [GRI06]. Quite on the contrary, in practice, testers mostly carry out only extensive *unit testing* [BIN99], [MEZ07].

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). Corriveau and Arnold have described at length elsewhere [ARN09-a] such an approach and its corresponding tool, the Validation Framework (hereafter VF [ARN09-b]).

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 will be illustrated in the next subsection. ACL is closely tied to requirements by defining syntax/semantics for the

representation of scenarios, and design-by-contract constructs [MEY92] such as pre and post-conditions, and invariants (rooted in [HELM90] and [HOL92]).

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 next subsection) 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 [ARN09-b]. 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 (e.g. in [COR07-a], [COR07-b], and [BASH08]) 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. (Details lie beyond the scope of this dissertation and are available at [ARN09-a]).

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 [*Ibid.*] indicated by the TRM. The execution paths are used to determine if each scenario execution *matches* the grammar of responsibilities corresponding to it within the TRM (see next example). Next, metric evaluators are used to analyze and interpret any metric data that was gathered during execution of the IUT. All of the results generated from execution of the TRM against the candidate IUT are written to a Contract Evaluation Report (CER).

The generation of the CER completes the process of executing a TRM against a candidate IUT. The CER indicates whether the candidate IUT matches the TRM, and where any deviations from the TRM were observed. For example, when a pre- or post-condition fails, the execution proceeds but that failure is logged in the CER. Also, when a scenario is executed by an IUT, the specified grammar of responsibilities must hold in order for the scenario to be considered to have succeeded. That is, for success, the responsibilities that compose the scenario must be executed in an order that satisfies the grammar. If the scenario cannot be executed, or responsibilities/events that are not defined by the scenario are executed, then the scenario is deemed to have failed. This mismatch is also reported in the CER. See Figure 1.1 below for an example.

Several quality control and analysis methods are being developed by other members of our research group to be used to analyze the generated CER and apply their findings to the software development process, or calculate information important to management and other stakeholders.

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. As explained above, 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. Finally, beyond automatic instrumentation, work is in progress [COR11-a] to have the VF

generate automatically a test suite (which is to be automatically instrumented in the IUT).

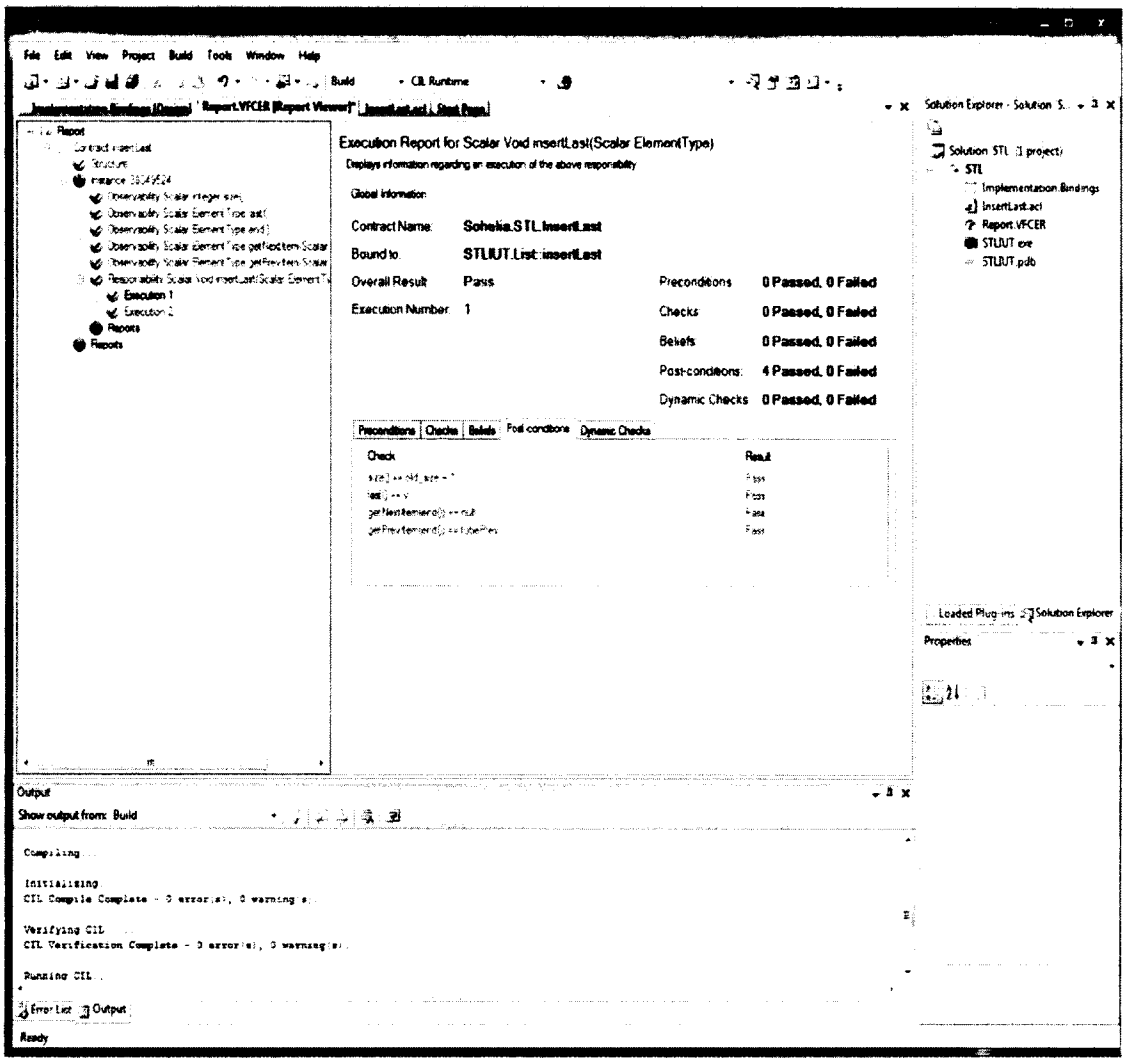

Figure 1.1 - A Contract Evaluation Report for a Single-linked List Member with C# IUT

The problem we address in this dissertation 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, through the walkthrough of a simple example in the next subsection.

1.1.2 An Example Contract

The following annotated example summarizes several of the semantics currently supported by the ACL contract language. The // and /* */ delimit comments aimed at explaining the key points of the example, which models a simple generic container (storing elements of some type tItem). We have included example usage of inheritance in order to demonstrate how our work supports the compositionality of contracts through this mechanism. Contract elements without comments are assumed to be self-explanatory.

```
Import Core;
Namespace My.Examples
{
/*An abstract contract is NOT bound to a type within the IUT. Also, T will be bound upon
ContainerBased being refined.  A contract may define variables, which will be kept by the VF.*/
abstract Contract ContainerBase<Type T>
{
 Scalar Integer size;   // Number of elements in the container.

/* An observability is a query-method that is used to provide state information about the IUT.
That is, they are read-only methods that acquire and return a value stored by the IUT.    */
 Observability Boolean  IsFull();
 Observability Boolean  IsEmpty();
 Observability T ItemAt(Integer index);
 Observability Integer   Size();
// An abstract observability MUST be refined in a derived contract.
abstract Observability Boolean  HasItem(T aItem);

/* The body of the "new" responsibility is executed immediately following the creation of a new
contract instance.    */
 Responsibility new()
{ size = 0;  Post(IsEmpty() == true); }
```

```
/* The body of the "finalize" responsibility is executed immediately before the destruction of the
current contract  instance. */
Responsibility finalize()
{ Pre(IsEmpty() == true); }

/* Invariants provide a way to specify a set of checks that are to be executed before and after
the execution of all bound responsibilities. Invariants precede pre-conditions, and follow post-
conditions. */
Invariant SizeCheck
{ Check(context.size >= 0);
  Check(context.size == Size()); }

/* This responsibility defines pre- and post- conditions for any addition. It is not to be bound but
rather to be extended by actual responsibilities.
The keyword 'Execute' indicates where execution occurs. */
Responsibility GenericAddition(T aItem)
{ Pre(aItem not= null);
  Pre(IsFull() == false);
  Execute();
  size = size + 1;
 Post(HasItem(aItem)); }

/* This responsibility extends GenericAddition. It therefore reuses the pre- and post-conditions
of GenericAddition. It does not add any other checks to those of GenericAddition.
But Add can (and will) be refined in the contract that extends the current abstract one. */
Responsibility Add(T aItem) extends GenericAddition(aItem)
{ Execute(); }

/* Insert also extends GenericAddition and thus reuses its pre- and post-conditions.
But it also adds pre- and post-conditions of its own due to the fact that its interface involves the
use of an index. */
Responsibility Insert(Integer index, T aItem) extends GenericAddition(aItem)
{ Pre(index >= 0);
  Execute();
  Post(ItemAt(index) == aItem); }

/* Responsibility Remove returns the element removed.
The keyword 'value' denotes the return value.  */
Responsibility T Remove()
{ Pre(IsEmpty() == false);
  Execute();
  size = size - 1;
  Post(value not= null);
  Post(HasItem(value) == false); }
Responsibility RemoveElement(T aItem)
{ Pre(IsEmpty() == false);
 Pre(HasItem(aItem) == true);
 Execute();
```

```
 size = size - 1;
 Post(HasItem(aItem) == false); }

/* The following scenario merely consists of a trigger statement and a terminate statement.
There is no grammar of responsibilities between these two statements (in contrast to most
scenarios.) This scenario captures the fact that the addition of an element x must eventually be
followed by removal of x.  Here Add or Insert trigger the scenario, and Remove or
RemoveElement terminate it.
Notice the use of the 'dontcare' keyword for the first parameter of Insert.  */
Scenario AddAndRemove
{ once Scalar T x;
  Trigger(Add(x) | Insert(dontcare, x)),
  Terminate((x == Remove()) | (RemoveElement(x)));  }

} // End of contract ContainerBase

/* A TRM must include a main contract. It typically includes several other contracts.
The main contract of a TRM must be bound to a type of the IUT. Here Container inherits from
ContainerBase.
Single and multiple inheritance are supported for composing contracts together.
Also, note that T in ContainerBase is explicitly  bound here to the type tItem (using syntax similar
to templates in C++). */
MainContract Container extends ContainerBase<tItem>
{ List Integer container_times;
// Amount of time that each item spends in the container.
  Scalar Timer item_timer;
/* Timer is a built-in type of our VF
A single timer can be used to time multiple items concurrently. */
  Scalar Integer number_of_items;
/* Used to store the total number of items that are stored by the container during execution.  */

// The abstract responsibility of ContainerBase is now refined.
refine Observability Boolean HasItem(tItem item)
{ tItem x;  Boolean result = false;
  loop(0 to Size())
{ x = ItemAt(counter);
  result = result || x == item;     }
  value = result; // Value is the keyword for return value.      }
/* A parameter can be set explicitly, or  using the binding tool of section 4, or set at run-time.
Here, it controls whether the static check below is to be performed or not. */
Parameters
{ Scalar Boolean CheckMembers; }

/* What follows is a static check that uses the plug-in static check HasMemberOfType to verify if
the container holds instances of type tItem. This check is performed only if parameter
CheckMembers is true. A belief is merely a message logged in the report (CER) produced by the
VF. */
Structure
```

```
{ choice(Parameters.CheckMembers) == true
{ HasMemberOfType(tItem); }  }

/* We refine new: The post-conditions of the parent contract are checked before the body
below is executed. */
refine Responsibility new()
{  number_of_items = 0;
   container_times.Init(); }

/* The 'fire' keyword is used to create an instance of an event that can, in turn, trigger or be
observed in scenarios.  */
refine Responsibility finalize()
{  fire(ContainerDone); }

/* Next, Add, Insert, Remove and RemoveElement from the ContainerBase contract are further
refined to use timers.
More specifically, the scenario AddAndRemove (in the parent contract) creates an instance of
itself for each element that is added to the container. This allows us to start a timer in Add or
Insert upon insertion of an element and to stop that timer when that element is removed. In
turn, this allows us to store the time spent by an element in the container.  */

refine Responsibility Add(tItem item)
{ Pre(HasItem(item) == false);
   Execute();
   item_timer.Start(item);  // Built-in way to start a timer.
  number_of_items = number_of_items + 1; }

refine Responsibility Insert(Integer index, tItem item)
{ Pre(HasItem(item) == false);
   Execute();
   item_timer.Start(item);
  number_of_items = number_of_items + 1;  }

refine Responsibility tItem Remove()
{ Execute();
  item_timer.Stop(value);
  container_times.Add(item_timer.Value(value)); }

refine Responsibility RemoveElement(tItem item)
{ Execute();
   item_timer.Stop(item);
   container_times.Add(item_timer.Value(item)); }

/* This responsibility is to be used in the scenario ContainerLifetime below. RemoveScn
abstracts away which of the two Remove responsibilities is used.
Notice again the use of keyword 'dontcare'.  */
Responsibility RemoveScn()
{ Remove() | RemoveElement(dontcare); }
```

```
/* A stub responsibility is a place holder for one or more responsibilities. Here, we have only one
choice, the default one, the  responsibility Add.  Parameters and other mechanisms could be
used to select between different kinds of addition, as illustrated elsewhere [15]. */
stub Responsibility AddElement(tItem item)
{ Pre(item not= null);
  [Default] Add(item); }

/* This scenario illustrates a Trigger being followed by a grammar of responsibilities and then a
Terminate statement. In this case, the Terminate MUST be preceded by an 'observe' statement
specifying the event that enables this termination.

In the following scenario, a new scenario instance is created each time a new container is
constructed (via the new responsibility). The responsibility new acts as the trigger.
The ',' denotes the 'follow' operator.
An atomic block defines a grammar of responsibilities so that no other responsibilities of this
contract instance are allowed to execute except the ones specified within the grammar.
The scenario must observe the event ContainerDone before concluding by proceeding with the
execution of  finalize (which fires the event ContainerDone before its checks. This semantic
'contortion' is due to the way scenario instances are monitored. */

Scenario ContainerLifetime
{ Trigger(new()),
  atomic
    { (Add(dontcare) | Insert(dontcare, dontcare))*,
(RemoveScn())*; },
  observe(ContainerDone),
  Terminate(finalize()); }

/* A list of integers representing the amount of time that each element spent in our container.
*/
Metric List Integer ContainerTimes()
{ context.container_times; }

// Total number of items that were stored in the container.
Metric Scalar Integer NumberOfItems()
{ context.number_of_items; }

// This section builds the evaluation report.
Reports
// {0} is where the reported result goes in the output string.
{ Report(     "The average time in the container is {0} milliseconds",
      AvgMetric(ContainerTimes())); // Plug-in (ie user-defined procedure) AvgMetric

/* A report all statement performs the exact same way as the report statement, except that it
generates a single result for all contract instances.   */
  ReportAll("The average time in all containers is {0} milliseconds",
      AvgMetric(ContainerTimes()));
```

```
Report("The number of items added to the container is {0}",    NumberOfItems());
ReportAll("The number of items added to all containers is {0}", NumberOfItems());          }

/* The type tItem used for the elements of the container cannot be type bound to the container
nor any of its descendants. So, here, we do not allow lists of lists. */
Exports
{ Type tItem conforms Item
{ not context; not derived context; }    }
}}
```

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 responsibility and scenario failure). This may mislead some readers to believe ACL already handles some form of variability. In fact, it does not: all the IUTs that can bind to this TRM can do so 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 and bindings required by this TRM. So the question remains: how can ACL (and VF) be augmented to support variability?

1.2 The Proposed Solution

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 imposed on us: neither the ACL nor the VF can be modified[1]. That is, given the ACL/VF is an experimental tool of over 250,000 lines of code, which is still undergoing tests, it was decided by the head of our research group, professor Corriveau, that our goal to support variability in ACL contracts had to be achieved *without* altering the syntax or semantics of ACL, or the working of the ACL compiler, or the modus operandi of the VF.

[1] ACL grammar contains about 100 formal rules and the same number of mapping instructions (from ACL constructs to Contract Intermediate Language). Also ACL compiler and its validation tool VF contain more than 250,000 lines of code (more details can be found at [ARN09-a, ARN09-b]). As it can be seen modifying ACL constructs, grammar rules, mapping instructions, compiler and VF software is really a complex and time consuming task, which is out of the scope of this dissertation. Therefore in this research, we propose a framework to model and resolve variability for an ACL-based domain contracts without any changes to ACL/VF constructs and tools.

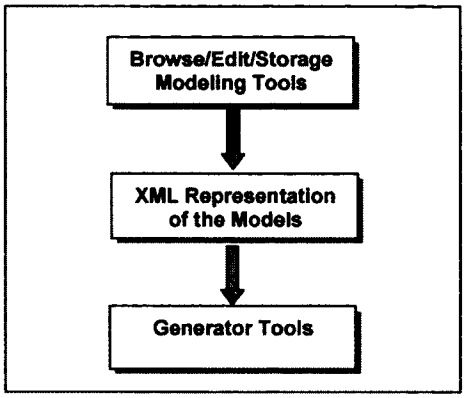

Figure1.2 - Cleaveland's Generative Approach

To do so, we adopt Cleaveland's [CLEA01] template-based general approach to variability, captured in Figure 1.2.

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 latter representation that is used to generate a specific member of the domain. Cleaveland elaborates: "XML is a standard data representation language. It is a reasonable language for representing models and achieves separation of concerns [between domain engineering and application engineering, which is highly desirable according to this author]. It is actually more than that because it integrates the system with the much wider world of XML technologies and tools, many of which are already suitable for generating other software artifacts. For example, XSLT is a standard transformation language for transforming between XML languages or to other text-based languages. Thus, by adopting XML as the interface, it is possible to readily use off-the-shelve standard components for creating generators."

Cleaveland's approach to generation is a *template-based* one rooted in the use of XML-based tools. Czarnecki [CZA02] explain (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". In contrast, a programming language based generative approach, such as Czarnecki and Eisenecker's [CZA00] C++ metaprogramming, typically uses advanced programming techniques (such as partial template specialization in C++ [ALE01]) that are 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 method we propose for tackling variability in ACL[2], which is illustrated in Figure 1.3, indeed adopts a template-based approach rooted in XML.

While the details of this figure are discussed at length in chapter 3, an overview of our proposed solution is useful at this point in order to understand what falls within the scope of this dissertation and what does not.

[2] This should be noted that while modeling hierarchy and inheritance are supported by ACL constructs, but these constructs are not sufficient to model variability contracts in ACL. The reason is because that using inheritance will lead to one sub-class per possible member contract, and therefore for a huge number of member contracts for a domain this will end up to an explosion of sub-contracts that can simply overlap. Plus the point that: for any new variability a new sub-contract should be specified from the scratch. The concept of huge number of sub-contracts (not generating), their overlapping (not reusing), and writing sub-contracts from scratch (not using configuration list) are completey on the contrary with the concept of generative process for a domain.

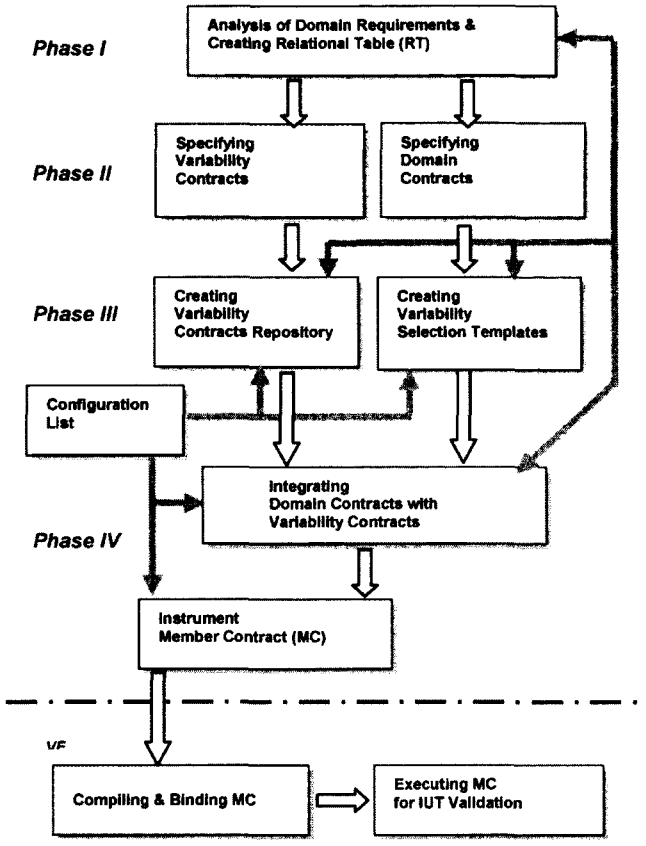

Figure 1.3 - An Overview on our Generative Approach

Domain engineering starts, like most existing approaches to system family engineering and software product lines with feature modeling. This first phase involves the (manual) creation of a *feature diagram*, as well as of a *feature grammar* (called here a variability *relationship table*), which addresses relationships between features [CZA00]. These two artifacts target variability in the domain. In Phase II of our approach, the commonalities of the domain at hand are captured (manually) in ACL contracts forming the *domain contracts*. In the same phase, the feature diagram and relationship table of phase I evolve into *variability contracts* (via algorithms we have developed for this specific purpose). These latter contracts are written using new syntax we developed for this purpose. The idea is to capture how each variant of a variation point affects the domain contract. Without going into details, a portion of such a variability contract is

given in Figure 1.4. Intuitively, feature VP1 addresses whether a container type has fixed or variable length (captured using the two variants "Variable" and "Fixed"): if length is variable, then the *isFull* observability necessarily returns false, otherwise it returns whether the actual size has reached the maximum fixed size. Since the semantics of such variability contracts do pertain to concepts of ACL, we have named the language used to capture variability contracts ACL-V (for ACL with variability).

```
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 1.4 - A Variablity Contract

It must be emphasized that there is a direct correspondence between the features and variants identified in the feature diagram of Phase I, and these variability contracts. Conceptually, variability contracts define the 'actions' to be performed on the template (i.e., the domain contracts in our work) when a particular feature value is present in the configuration input to generate a specific member of the domain. Such actions can be carried out by a template processor only if the domain contracts define where they are to take place. 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. Furthermore, both the domain and the variability contracts must be transformed into their XML [XML-a], [XML-b]

equivalents in order to be made usable by the template processor. This is what Phase III of our approach tackles:

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

b) Everywhere in the domain contracts where variability can modify these contracts, a *variability selection template* (which can be thought of as a placeholder for the ACL 'code' to be plugged upon the selection of some action from a variability contract) is inserted. (In our approach, given our specific approach to template processing, these *variability selection template* take the form of XSLT style sheets [XSL-a], [XSL-b], [ONO02], as will be explained in chapter 3.)

At the end of phase III, both the variability contracts and the domain model have been transformed into artifacts that serve as input to the generator, which also requires a configuration list specifying the exact list of feature values to be used to generate a specific member of the domain, which we call a member contract. Phase IV of our approach deals with the generation proper of such a member contract, that is, of an ACL contract corresponding to the input configuration list. This ACL contract can then serve as input to ACL/VF so that it can be compiled and tested. Phase IV is fully automated and thus its modus operandi is of no import at this point of the dissertation.

It must be emphasized that, in contrast to many existing generative approaches, we not only define the artifacts relevant to the generative process but, most importantly, we define algorithms to go from the more abstracts one to those directly used by the generator. That is, traceability between all the artifacts of our approach is critical. So is the use of a feature grammar: relying on the user to supply only valid configurations lists is a gross oversimplification that we avoid. The development of algorithms for feature grammar enforcement, traceability and generation corresponds to the most significant investment of time in this work, their testing coming in second place. Now that these algorithms have been systematically verified (through two extensive case studies),

another student of our research group has been assigned the task of a) creating user-friendly interfaces to input the domain and variability contracts and b) implementing our traceability algorithms. The point to be grasped is that while this dissertation presents Phases I, II and II as manual ones, there is no reason to prevent some form of automated support for these phases, but this lies out of the scope of this dissertation.

It must also be stressed that our approach to capturing variabilities and commonalities of a domain using ACL contracts does not depend on the semantic 'correctness' of the feature diagram, feature grammar and domain contracts supplied by a user. (In fact, as Arnold explains at length in his dissertation [ARN09-a], enforcing the notion of correctness in the context of specifications by contracts is quite problematic.)

Finally, an attentive reader will eventually notice that not all semantic elements of ACL have been considered for potential variability. This decision by professor Corriveau proceeds from the experimental nature of the ACL/VF at this point in time. For example, scenario monitoring by the VF, as well as support for non-scalars, is still not stable enough in the current version of the tool to warrant consideration for variability. More precisely: a key facet of our approach to the validation of our work rests on the ability of the ACL/VF to compile and execute the generated member contract for extensive testing. This is only possible if we restrict variability to those elements of ACL that can indeed be tested deterministically!

The validation of our proposed solution for the generation of an ACL member contract from a domain contracts, variability contracts and a configuration specific to that member has been carried out as is usually the case in domain engineering and application engineering (if not in software engineering in general), by developing two extensive case studies. Both case studies 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 pertains specifically to sequential containers (such as arrays, lists, stacks and queues). This choice was straightforward given that our research group had carried out earlier work [TIA05] on feature modeling

across a large set of such container libraries (including those found in the Standard Template Library (STL of C++) [STL95] and that Czarnecki and Eisenecker [CZA00] had also presented a detailed discussion of the domain engineering of lists. In other words, we wanted to avoid the all-too-frequent 'toy' example in favor of a concrete example. Indeed, this case study allowed us to generate member contracts that were bound to an STL-like library in managed C++. 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 to be grasped 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 at the expense of being grounded in the complexities of actual industrial code. The point must be made however that, despite the similarities between these two case studies, our proposed solution is domain independent. Indeed, our approach is being applied by other members of our research group to two other vastly different domains:

a) the modeling of queue management in grocery stores

b) the modeling of registration management in universities

Through these experiments we hope to confirm the generality of our proposal.

1.3 Contributions

This dissertation can be viewed as a study of how feasible it is to implement an XML template-based generative approach for a requirements specification language for model-based testing. While there is a multitude of such specification languages (see the second chapter of Arnold's dissertation [ARN09-a]), ACL stands out as the only such language we know that is *not* rooted in the use of state machines (thus avoiding the state explosion problem explained at length in [GRI06]) and yet generates an instrumented IUT *and* executable tests. Also, to the best of our knowledge, we believe that despite design by contract [MEY92] dating back to 1986, our work is the only one to attempt to introduce variability to *domain contracts*. The choice of ACL as the language

'on top of which' to investigate variability is further motivated by the fact that it readily offers a method for the validation of what is generated: the member that is generated is a *bona fide* ACL contract and thus, if it is correct, can be input to ACL/VF, compiled and run. As a matter of fact, figure 1.1 is an example of such a generated contract (for a single linked list) having been compiled and run against an implementation under test.

In summary, the main contribution of this work is a domain- and language-independent generative process we propose for generating ACL member contracts from ACL-based domain contracts. This process is comprehensive inasmuch as it addresses how the two traditional artifacts of domain engineering (namely a feature diagram and a feature grammar [CZA00]) 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).

Several other contributions proceed from this main one:

1) Our work contributes to illustrating the power and generality of the template-based approach to generation advocated by Cleaveland [CLEA01].

2) Our proposed process emphasizes the importance of traceability [COR96] in any model-driven process involving several model transformations. The proposed traceability links, provided in section 6.2.1, is the base for our future work (section 8.2).

3) The two extensive case studies we developed not only provide strong evidence for the validation of our work but also demonstrate the feasibility of domain contracts with variability for off-the-shelf components such as libraries of containers. (In this dissertation we focus on STL-inspired [STL95] containers.)

4) Further systematicity is obtained by having each transformation (from one model to another) that we use be defined by an algorithm (all of which are being investigated by another student of our research group who is to implement them).

5) In the course of choosing a template-based approach to generation, we studied at length existing approaches to modeling variability, as well as model-driven generative approaches. Our findings are reported in chapter 2.

1.4 Plan for Dissertation

In the rest of this dissertation, we elaborate on the domain- and language-independent solution we propose for generating ACL member contracts from ACL-based domain contracts. We first review in chapter 2 a) model-driven transformational frameworks and tools that do not directly support variability, b) approaches to modeling variability in a domain and c) model-driven generative approaches. Then, in chapters 3 through 6 we study at length the complete generative process by following the development of our first case study. Chapter 3 tackles the creation of the feature diagram and, most importantly, of the feature grammar. Then, in chapter 4, we discuss how to go from these two artifacts to domain contracts (capturing commonalities inherent to this domain) and variability contracts (addressing how variation points and their variants affect the domain contracts). Chapter 5 focuses on how our template-based approach relies on xml/xslt 'equivalents' of domain and variability contracts. Finally, in chapter 6, we discuss how, once a member contract has been generated, it can be validated using the ACL/VF. In chapter 7, we summarize our second extensive case study. We conclude the dissertation with a summary of our results and a brief overview of future work. The appendices hold the complete listing of actual files that served as input or we generated as output of our process.

2 Literature Review

This chapter will review existing research in three areas: 1) model-driven transformational frameworks and tools that do not directly support variability, 2) approaches to modeling variability in a domain and 3) model-driven generative approaches. The chapter is structured along those three areas in order. Our review is summarized in two tables: Table 2.1 (in subsection 2.2.3) for the research in the first two areas and Table 2.2 (in subsection 2.3.2) for the work in the last area (a summary of this chapter can be found in [BASH11]).

2.1 Model-driven Transformation Frameworks and Tools

In this section, we offer a brief summary for several model-driven transformational frameworks and tools. We then compare, in Part (a) of Table 2.1, these contributions with respect to their compliance with modelling standards such as Object Management Group's [OMG] Meta Object Facility [MOF] and the Eclipse Modeling Framework [EMF] and their support for a) modeling variability, b) traceability, c) and executability of the generated member of the domain.

2.1.1 Related Work

2.1.1.1 Blouin et al.

Model-driven software engineering promotes the use of transformations between models (e.g., at different levels of abstractions), like in [deMI05]. Transformations are not the same things as mappings. For example, Blouin et al. [BLOU07] argued that using XSLT [XSL-a], [XSL-b] to transform XML [XML-a], [XML-b] documents has become the norm for model transformation. But, to these authors, this is problematic because writing XSLT scripts requires substantial expertise. Consequently, they suggested a simpler mechanism for obtaining a model from another, namely mappings. This method consists in: a) first defining a mapping relation between the source and target schemas, expressed in some mapping language, and then b) using a transformation process that applies the mapping to instances of the input model. Their claim is that is a much

simpler approach than defining transformations. In fact, this claim is controversial. For example, Madhaven et al [MAD02] demonstrated that "generating mappings is a labour intensive and error prone task" involving complex inference. For these authors, the generation of mappings requires well-defined semantics supporting reasoning and comparison between mappings. To this end, they proposed a framework for defining languages for specifying mappings and for specifying the semantics of those mappings

2.1.1.2 Tefkat

In [LAW05] the authors introduced Tefkat, a modeling transformation tool that, like ATL [ATL], was built on top of EMF. The tool provides a declarative, logic-based language that can be used effectively to specify a set of constraints – in the form of assertions – that should hold over the source and the target models. It also supports traceability by tracking classes between the source and target models. That is, as in ATL, such traceability must be defined explicitly.

2.1.1.3 mSyntra

In [IVK04] and [IVK05], the authors proposed an approach to synchronize software artifacts across levels of abstraction. In their approach, model dependencies are implicitly encoded using transformation rules, and when two models are synchronized, they are evaluated using an equivalence relation.

2.1.1.4 MTF

The Model Transformation Framework (MTF) [MTF] is a transformation tool provided by IBM based on EMF. It implements some of the QVT [QVT] concepts and provides a simple declarative language that can define mappings between the source and the target models. In other words, the MTF transformation is achieved through a set of mappings that relate items between the source and the target models. As the mappings can be bi-directional, each direction should be defined when the transformation is invoked. However, this bi-directionality puts lots of constraints on

both models which further limits accessing the items in both models, and hence some of the transformations cannot be implemented at all.

2.1.1.5 UMT

UMT [UMT] is the UML Model Transformation Tool [UML], designed to support a) Text-to-UML transformation for systems that don't have UML-based models (such as legacy systems), b) UML-to-UML for refinements/transformations among UML-based models (such as PIM-to-PSM for software radio components [OMG06]), and c) UML-to-Text transformation for generating code and documentation. Drawbacks of UMT [GRON05] include: a) UMT is not based on a meta-model and therefore it is not MOF-compliant, b) UMT needs an extra transformation to/from XMI [XMI] light, and c) as for ATL and Tefkat, in UMT the traces between the source and the target model elements must be implemented explicitly.

2.1.1.6 MTRANS

According to the MOF meta-data architecture [MOF], a transformation between a source and a target model can be either at the model level (M1) or at the meta-model level (M2). For example, in the work of MTRANS [PEL01], [DOM00], and [GER02], the transformation is performed at M2 level, whereas in [POL02] and [GAL05] the transformation is at the M1 level. Let us briefly elaborate on such work.

In [PEL01], the authors introduced MTRANS as a general framework to express model transformations at the level of meta-models (M2) for all MOF-compliant models. In the first version of their work, they used XSLT to perform the transformation automatically from a source model to the target model, which are both expressed in XMI. To avoid the direct specification of XSLT templates, in the next version of their work they introduced the MTRANS language. The code written in MTRANS defines the transformation rules from source to target meta-model, which in turn will be compiled into their corresponding XSLT code.

In [DOM00] the authors proposed an informal description of the transformation process, while in [GER02] the authors provided a transformation method using XSLT and logic programming. In [POL02] the definition of transformation rules is based on an extended OCL [OCL], and in [GAL05] the authors integrated model-checking tools with model-analysis techniques to obtain the target PSM model using MDA methods as well as the XML language.

In [PEL01] and [KUR02], the authors explored the possibility of exporting a model transformation from the MDA [MDA] to the XML, and to do so they proposed XSLT processing on the XMI serialized format of the input model.

Finally, in [BEZ03-a] and [BEZ03-b], the authors considered the emergence of the OMG MOF/QVT [QVT] standard and the existence of model transformation tools such as ATL [ATL]. According to the authors, who considered various transformation environments in the context of the MDA, XML is a better choice. In their opinion, XML offers as much functionality as MDA tools but provides better traceability and more flexibility with respect to the transformation process itself. Thus, ultimately in their work, models captured as XML documents and required transformation rules are assembled into an XSLT file that is then transformed into a XQuery [XQU] file as the output. This approach is conceptually close to our proposal, with the difference that our output is not XQuery but an actual ACL contract.

2.1.1.7 E-MORF

In [GER03-a], [GER03-b] and [DUD03-a], [DUD03-b], [DUD04], the authors discuss at length the fundamental differences between the MOF and the EMF modeling frameworks and offer mappings between the two systems. In [GER03-a] and [GER03-b], the authors proposed the E-MORF tool and framework to transform between MOF- and EMF-based models instances. The proposed framework starts by defining of the initial MOF or EMF models in XMI format. As the initial models are different, their MOF-XMI and EMF-XMI specifications are not interchangeable. Next two XSLT scripts are

developed that contain the mapping rules between MOF-XMI and EMF-XMI models and their instances respectively. For example in the MOF environment, by applying MOF-to-EMF XSLT transformations to MOF-XMI and MOF-XMI-instances, these can be transformed into their corresponding files in EMF environment. The reverse process can be done in EMF environment by applying the relevant EMF-to-MOF XSLT transformations. As some data types cannot be modeled in EMF, they are modeled as Java data types. Therefore in transforming from EMF into MOF, the authors used JavaML [JAVA-b] to translate Java data types into XML. Also in their framework, while the first XSLT scripts that transform between MOF and EMF based models are written manually, the second set of XSLT scripts that transform instances of such models are generated automatically.

2.1.1.8 JMI

The Java metadata interface (JMI) [JMI] provides a mapping between MOF1.4 and Java. The JMI specification provides a platform-independent infrastructure for the creation, storing, retrieving, and interchange of metadata. The JMI implementation provides the possibility of generating a Java interface for each MOF-based meta-model. It also supports the interchange of metadata and meta-models through XML documents with XMI format.

2.1.1.9 AMOF2.0forJava

The aMOF2.0forJava [JAVA-a] provides a mapping between MOF2.0 and Java. The main idea behind this tool is similar to JMI but for MOF2.0 instead of MOF1.4. Roughly put, the goal is to capture the new MOF-based modeling capabilities and to provide these capabilities to programming environments with Java. They explain: "In contrast to the earlier MOF versions 1.x, where abstraction hierarchies were usually rooted in a single "Element", leading to a limited modeling of abstractions, the new UML/MOF2.0 uses multiple abstractions, each covering a simple aspect of data-modeling, as a library for itself and your meta-modeling tasks". This tool is a meta-modeling tool based on UML/MOF2.0 and on CMOF [MOF] that provides support for type safe meta-data

handling. In particular, this tool is able of a) generating repositories of meta-models, b) interchanging meta-data with models and meta-models in XMI format [XMI-a], c) offering an implementation of CMOF models, and d) providing an implementation of UML library based on CMOF.

Both of JMI and aMOF2.0forJava strive to facilitate creating and handling of a repository of meta-models and meta-data in a way that the repositories can be analyzed and transformed by other modeling tools. In our research however, neither the analysis of such meta-models repository nor their mapping to programming environments are relevant goals. Thus such tools are of Therefore the relevance of these tools to our research is rather moot.

2.1.1.10 FeatureMapper

In [HEID07] and [HEID08-a] the authors proposed a tool called "FeatureMapper" that maps features from a feature model, to their corresponding artifacts that realize those features in an SPL. The tool selects features from the feature model according to a configuration model (in problem space), and then eliminates those elements that are mapped to non-selected features (in the solution space), to derive a product model. It is similar to the work of Czarnecki in [CZA05-c] to map features to the other models' artifacts. In [HEID08-b] the authors proposed a visualization technique called "MappingView" to provide four different visualization views, where in each view different properties of SPL can be represented that were hidden before. This visualization technique is implemented in "FeatureMapper" too. Then in [HEID09] the authors instrument the FeatureMapper to enforce constraints on feature models and configuration models to ensure well-formedness of all SPL models (having valid models, e.g. in [CZA06]). Finally in [ZSCH09] the authors discussed about VML*, a customized model transformation tool, where in [HEID10] they provided a comparison between their FeatureMapper and VML*.

The authors stated that as VML* is based on customizing a language to a target modeling language, therefore it needs a setup cost and a mapping specification for each transformation. Also the authors didn't mention if any changes happen to the model, is it possible to trace back the mappings to the previously selected features or not?

2.1.1.11 ATL

The ATLAS [ATLAS] Transformation Language (ATL) [ATL] is a language proposed by the Object Management Group (OMG) [OMG] to perform general transformations within the MDA framework. Due to its increased visibility (stemming from being put forth by OMG), several of its shortcomings have been identified, categorized as follows:

a) ATL is a model transformation language [ATL] that provides essential constructs for a descriptive and declarative model specification; therefore it is not suitable for modeling with validation purposes.

b) ATL performs a transformation based on some pre-defined transformation rules [BEZ03-a], [BEZ03-b], but it doesn't have any mechanism to organize these rules.

c) In ATL, the traceability model must be generated explicitly [JOU05].

d) In ATL, transformation rules are defined using OCL [OCL] expressions, which considerably restrict the semantics of such rules.

e) ATL is specific to EMF [EMF] and Eclipse [ECL] environment.

f) ATL transformation rules map the artifacts from the source model to the target model, and hence for each new target model a set of new rules should be defined. While in generative process, a configuration list [CZA00] defines target domain member.

2.1.2 Discussion

Ultimately, like all other transformational technologies, ATL does not offer generation per se. This is a crucial point to emphasize. A truly generative approach

necessarily involves the use of a feature grammar and of a configuration corresponding to the domain member to generate. While transformational approaches are currently very popular, within the specific context of modeling variability, they are in fact semantically more constrained than truly generative approaches. As emphasized by Czarnecki and Eisenecker [CZA00], a generator assembles a solution from reusable assets. This assembling depends on a grammar defined in terms of configurations of features and feature values. Conversely, transformation rules focus on how individual elements of a source model can be mapped onto elements of the target model. Semantically, such rules are 'local' to specific elements of the source model, in contrast to the more 'global' rules of a feature grammar, which tackle, we repeat, configurations of features. Thus, transformational approaches appear to be less suitable for domain and application engineering than generative methods. This conclusion partially explains our choice of XML technology for our proposed solution (the choice of a template-based method being another crucial consideration). With respect to ATL in particular, the fact that [BEZ03-a], [BEZ03-b] had to augment this language to make it more suitable for their purposes further convinced us of not using this technology: while an XML-based approach to generation offered both unconstrained semantics and a ready-to-use generator (XSLT), ATL and similar transformational approaches did not.

2.1.3 Summary Table

Table 2.1 Part(a) summarizes the above discussion (see 2.2.3).

2.2 Modeling Variability with and without Meta-Modeling Support

2.2.1 Related Work

In this section we briefly discuss research in the area of variability modeling (which is discussed at length in [BOSCH02], [BOSCH00], [SAV00], [SAV02], [POH06], [THI02-a], [THI02-b], [vanG01], [VIS03], [MYL02], [MYLO01], [KEE99], [JARI04], [McG05], [McG07], and [Claußθ1]). We describe the major characteristics of each selected approach. A comparison between these approaches can be found in Table 2.1 Part (b).

In [BUH04] the authors argued why it is not sufficient to model requirement variability with a feature model such as (FODA) [KAN90]. For example in a feature diagram, it is not possible to define the relationships between the variation points of a domain: a feature grammar is required. In [BUH05] the authors proposed how to model variability within the requirements and how to present the dependencies among the requirements through an extensive study. However they didn't provide the details of how to derive a member from this variability model.

In [MOR08] and [VIN07], the authors proposed to integrate requirements variability within requirements meta-model, the Requirement Engineering Meta-Model (REMM). They defined a catalogue of requirements in their REMM meta-model and stated that this is a catalogue for both defining requirements "for reuse" and creating a member from it "by reuse". They also introduced REMM-Studio, a graphical-based tool implemented to support requirements modeling, reuse and validation. Their work is similar to ours in its goal of defining a 'catalogue' of requirements for reuse. However, they do not separate between commonalities and variabilities. Also, their catalogue is captured at the meta-model level whereas our requirements (in the form of ACL contracts) are tackled at the model level.

[MOO07] proposes two separate meta-models for representing domain requirements and domain architecture. The domain architecture meta-model reflects the OMG Reusable Asset Specification (RAS) and adopts a layered meta-modeling structure. The authors first classify the types of variation points at the architectural level and then explicitly represent them in the domain architecture model. In both meta-models, they address variability and the traceability between the domain artifacts.

In [BAC03] the authors proposed a high level meta-model for representing variability in a software product line. In this meta-model, their main focus was separating the representation of variability from the representation of different artifacts of the domain while these artifacts are being generated. Meanwhile the traceability between the variation points and these artifacts was also being maintained.

In [BER05] the authors proposed a conceptual variability model, rather than a meta-model, that contains a traceability mechanism to trace variation points at different levels of abstraction and across different generic artifacts (mostly requirements) of a software product line. In order to implement this conceptual model, they represented variability separately, so that the variation points could be linked and traced back to those generic artifacts.

In [GOM02], [GOM04-a], [GOM04-b], and [WEB04] the authors proposed a multiple view meta-modeling approach for variability management in a given domain. Their work clearly shows the relationship between the goals of modeling variability and the specific approaches for variability management. For example, if the goal is to model variability at architectural level, inheritance can be used. Using the other variability modeling approaches like parameterization is not suitable at the architectural level since parameterization is completely related to the implementation artifacts.

2.2.2 Discussion

Recently some attempts have been made at defining a 'standard' variability model. For example, in [POH05], [BUH03], [BUH05] and [LAU05] the authors introduced their "Orthogonal Variability Model" (OVM), which they claimed is different from the existing ones. They provided a collection of common rules for guiding the design and implementation of artifacts of a software application, and then how to combine these artifacts to generate that application. They claimed that during this generation process variability can be described by OVM in different models such as feature models, use case models, component and test models, where a design selection has to be made, without modifying the existing models' notations. For them, variability in the target application's architecture can be represented by refining the orthogonal variability model and adding internal variability – only visible for the engineers – to it. In [BUH05] they proposed a variability meta-model which is implemented in tool DOORS [DOORS]. This tool is used to represent variability among SPL requirements.

In [HU04], [HAU08] and [BAY06] the authors proposed the "Consolidated Variability Model" (CVM), linked to MDA [MDA] standards, that provides descriptive techniques for representing variants of UML state diagrams or variants of UML sequence diagrams. These diagrams have been enhanced with annotations to represent variation points. The authors also wanted to show that all variants do not have to be anticipated when modeling the domain, even at the presence time of the variants. More specifically in their work, the "product line model" is represented using UML2.0 [UML2] use cases, and the "system family model" is represented using UML2.0 composite structures. In the latter model, which is equivalent to a Platform Independent Model (PIM), variability is represented through stereotypes. To link variability requirements to system family model a model transformation is used. In their derivation approach, first a "product model" is defined similar to a "product line model". Then the "product model" will be linked into the "system family model" through a transformation process. The result of this transformation, called "product/system model" is correspondent to the PIM model of the product. Finally after more transformations on this result, the target product is realized in the target platform.

But a 'standard' variability model in the area of model-driven engineering does not currently exist. While in OVM the authors try to document all the traceability links between their variability model and the feature model artifacts, in CVM the authors try to minimize the number of stereotypes they use to represent variability and instead using the built-in mechanisms of UML. Moreover, none of the above two methods are based on UML meta-models, and therefore they need to map their variability models to UML diagrams. And there is no discussion about their mapping techniques.

In [GRO07-b] the authors provided a variability model based on aspect-oriented modeling techniques, and in [VOE07] we find a variability implementation in model-to-model transformations and code generation using aspect-oriented model-driven approaches [GRO07-a]. That work supports integration with different tools such as [Pure] as a variant manager and [oAW] as a model-driven software development

[STA06] toolkit within the Eclipse environment. The resulting models can benefit from built-in support for EMF-based meta-models. Their work is somewhat similar to our work in trying to integrate model-driven development with a generative approach for software product lines, but they rely on aspect-oriented modeling techniques, which are out of the scope of this research. It is our understanding however that aspect weaving is similar but semantic quite simpler than the use of feature grammars.

2.2.3 Summary Table

Table 2.1 summarizes the discussion above. The columns of this table are:

- Dependency to Modeling Languages and Tools.
- MOF/EMF [MOF], [EMF] Compliance
- Availability of support for variability
- Availability of support for traceability between models
- Availability of support for executability: can the generated domain member be traced to executable code
- Summary of Transformation Steps for modeling variability

The results of our investigation are categorized in two parts:

a) The frameworks and tools that do not directly support variability modeling for a domain. In these frameworks, domain variability can be modeled using other frameworks and tools. For example, if ATL [ATL] is used for model transformations, variability can be modeled using annotations in UML diagrams [UML] based on some variability modeling approach (such as in [GOM00], [GOM02], [GOM04-a], [GOM04-b]).

b) The frameworks and tools that directly support variability modeling for a domain. In these frameworks, domain variability can be modeled through pre-specified modeling rules, notations and approaches.

Table 2.1- Summary of the Survey on Modeling Frameworks, Tools and Modeling Variability

Modeling Frameworks, Tools & References Part (a)	Modeling Lang. & Tools Dependency	MOF/ EMF comp.	Var. Sup.	Trac. Sup.	Exec. Sup.	Summary Transformation Steps: from source model to target model
[BLOU07] [MAD02] [UML] [XML-a][XML-b] [XSL-a][XSL-b]	Mapping language, Strong reasoning, UML, XML, XSLT	None	No	No	No	-A framework for model transformation via mapping -Defining a relation between two schemas of source and target models -Applying the mapping on the instances of the schemas
Tefkat [LAW05] [DUD03-b] [DUD04] [GER02] [ECL][BUD03] [UML] [XML-a][XML-b]	Tefkat language, Eclipse, UML, XML	MOF/ EMF-based	No	No/ capture explicitly	No	-A logic-based tool for model transformation using patterns and rules -Specify a set of rules and patterns for constraints, tracking classes and transformation over the source and the target models using Tefkat -Using Tefkat engine on the top of Eclipse to generate the target model
mSynTra [IVK04] [IVK05] [UML] [XML-a][XML-b] [DTD]	UML, RUP, XML, DTD	MOF-based	No	No	No	-A framework to synchronize the system models at the abstract level -Using transformation rules to encode model dependencies -Transformations on one model are propagated to the other by model selecting possible paths that to maximize dependencies -Synchronized models are evaluated with equivalence relation
MTF [MTF] [ECL][BUD03] [UML] [XSD]	MTF Language, Eclipse, UML, Java, XSD	EMF-based	No	Yes	No	-A tool for model transformation based on EMF -Defining a set of mappings between EMF models in the Eclipse environment using MTF language -Mappings are bi-directional with lots of constraints
UMT [UMT] [QVT] [GRON05] [UML] [XMI] [XSL-a][XSL-b] [WAN02]	UML, XMI, XSLT, Java	None	No	No/ capture explicitly	Yes	-A tool for model transformation to support Text-to-UML, UML-to-UML, and UML-to-Text transformations only for UML-based models -Serializing model and meta-model into XMI -Generators are implemented in UMT environment, using XSLT or Java -Generating code and documentation via transformation to/from XMI
MTRANS [PEL01] [BEZ03-a] [BEZ03-b] [BEZ05] [DOM00] [GER02] [XMI] [ATL] [XSL-a][XSL-b]	MTRANS language, XMI, XSLT, ATL	MOF-based	No	No	No	-A framework for model transformation for all MOF-based models -Serializing model & meta-model to XMI -Defining transformation rules from source to target meta-models, using MTRANS language -Compiling transformation rules into XSLT format to define the target model
E-MORF [GER03-a] [GER03-b] [DUD03-a] [DUD03-b] [DUD04] [XML-a][XML-b] [XMI] [XSL-a][XSL-b] [Java-b] [ECL]	XML, XMI, XSLT, Eclipse, Java, JavaML	MOF/ EMF-based	No	No	No	-A framework for model transformation between MOF and EMF -Serializing model and model-instance into XMI in both MOF and EMF -Defining transformation rules from MOF-XMI to EMF-XMI meta-models and MOF to EMF model-instances and vice-versa using XSLT -Applying XSLT files on source model and instance to generate targets

Table 2.1- Summary of the Survey on Modeling Frameworks, Tools and Modeling Variability (cont.)

Modeling Frameworks, Tools & References Part (a)	**Modeling Lang. & Tools Dep.**	**MOF/ EMF comp.**	**Var. Sup.**	**Trac. Sup.**	**Exec. Sup.**	**Summary Transformation Steps: from source model to target model**
JMI [JMI] [XML-a][XML-b] [XMI]	XML, XMI, Java,	MOF-based	No	No	No	-A tool for meta-model and meta-data interchange by mapping Java &MOF1.4 -Specifying a Java interface using JMI to create, access and interchange with a MOF-based meta-model -Creating JMI implementation to handle an XML document defined in XMI
AMOF2.0forJava [JAVA-a] [XML-a][XML-b] [XMI] [UML]	XML, XMI, Java, UML	MOF-based	No	No	No	-A tool for meta-model and meta-data interchange through mapping between Java and MOF2.0 using UML & CMOF -Specifying a Java interface using this tool to create, access and interchange a MOF2.0-based meta-model with a Java programming environment
FeatureMapper [HEID07] [HEID08-a] [HEID08-b] [HEID09] [HEID10] [ZSCH09]VML*	FeatureMapper Mapping spec. Features	None	No	No	No	-A tool to transform features from feature model to solution space – arealized element – via a mapping specification -Selecting features according to a configuration model -Eliminating elements that are mapped to non-selected features in the solutionspace
ATL [ATLAS] [ATL] [JOU05] [ECL][BUD03] [UML]	ATL, Eclipse, UML	None	No	No/ addressed loosely	Yes	-A tool for model transformation from source model to target model -ATLAS transaction language (ATL) is a hybrid language: declarative and imperative constructs -Transformation model is expressed using ATL mapping rules
Part(b)						
[BUH04][BUH05] [XML-a][XML-b] [XSL-a][XSL-b] [ATL] [XMI] [UML] [DOORS]	DOORS, XML, XSLT, ATL, eXAct, UML	None Own-meta-model	Yes	Yes	No	-Modeling variability of a domain based on a meta-model -Managing variability using DOORS -Proposing a variability meta-model for structuring and recording the domain variability, dependencies & constraints among the variation points -Supporting the traceability between the variability model &requirements artifacts
[MOR08] [VIN07] [UML] [OCL]	REMM, REMM-Studio, UML, UML OCL, HTML	None Own-meta-model	Yes	Yes	No	-Modeling variability of a domain based on a meta-model -Proposing a Requirements Engineering Meta-Model (REMM) -Integrating req. variability within REMM -Defining a catalogue of requirements in REMM "for reuse" & design "by reuse"
[MOO07] [RAS] [UML]	RAS, UML	None Own-meta-model	Yes	Yes	No	-Modeling variability of a domain based on a meta-model -Proposing two meta-models for the domain requirements and the domain architecture and the traceability between them explicitly, based on OMG Reusable Asset Specification (RAS)
[BAC03]	–	None Own-meta-model	Yes	Yes	No	-Modeling variability of a domain based on a meta-model -Proposing a separate model for the domain variability from the model of other domain artifacts, and a traceability mechanism between them

Table 2.1- Summary of the Survey on Modeling Frameworks, Tools and Modeling Variability (cont.)

Modeling Frameworks, Tools & References Part (b)	Modeling Lang. & Tools Dep.	MOF/ EMF comp.	Var. Sup.	Trac. Sup.	Exec. Sup.	Summary Transformation Steps: from source model to target model
[BER05] [UML] [KAN90]	Diagrams, FODA, UML	None	Yes	Yes	No	-Modeling variability of a domain via a conceptual diagram -Proposing a traceability mechanism to keep the links between variation points and domain requirements
[GOM00][GOM02] [GOM04-a] [GOM04-b] [WEB04] [UML] [KAN90]	FODA, UML	None Own-meta-model	Yes	No	No	-Modeling variability of a domain based on a meta-model -Domain requirements are modeled using feature diagram and UML diagrams with annotations -Extending UML diagrams to represent multiple views of a product generation
OVM [POH05] [BUH03] [BUH05] [LAU05] [KAN90]	UML FODA DOORS	None	Yes	Yes	No	-Modeling variability of a domain using defined notation and meta-model -Domain requirements are modeled using UML diagrams with annotations -Mapping variability models with features, and -Integrating variability models in Domain
CVM [HAU04] [HAU08] [BAY06][KAN90] [UML] [UML2] [BERK03]	UML2.0 FODA	None	Yes	No	No	-Modeling variability using stereotypes -Modeling product line using use cases -Modeling system family using UML composite structures -Using a transformer with the last 2 models as input, deriving product model

2.3 Model-Driven Generative Approaches

2.3.1 Related Work

This section presents our survey of model-driven generative approaches that strive to generate a member from a domain requirements model. A summary is presented in Table 2.2.

2.3.1.1 XVCL

XML-based Variant Configuration Language or XVCL [XVCL], [JAR07], [JAR08] has its roots in Frame technology [EMR03], [XFRA]. It is a domain independent technique. In [JAR07], Jarzabek described how reuse of structural similarities extends the benefits of conventional component reuse. By conventional component reuse, he means reuse based on components at architectural-level, and by structural similarities, he means the repetition of patterns in software of any type, from similar code fragments to recurring

architectural-level components. He initially tackled reuse only for software programs [BASI05] and based on the principle of separation of concerns. He then made this idea concrete via the creation of a generative technique, the XML-based Variant Configuration Language (XVCL). He proposed that any repetition pattern in target systems can be represented with a generic adaptable XVCL meta-structure. He then developed, reused, and derived specific custom systems from their meta-level representations. He showed that the separated concerns can be parameterized with XVCL commands, in order to enhance a programmer's ability to define variations in code at any level and for any type that is required, from a subsystem or component level to a single program statement.

Frame Technology (FT) [FT] is a language-neutral system that manufactures custom software from reusable, machine-adaptable building blocks called frames. [BAS97] proposed the first FT in order to automate the repetitive, error-prone editing involved in adapting generated and hand-written programs to changing requirements and contexts. Formally, a frame is a procedural macro consisting of frame-texts – zero or more lines of ordinary program or text – and frame commands that are carried out by FT's frame processor as it manufactures custom programs. (Such frames should not be confused with frames in Artificial Intelligence!) Each frame is both a generic component in a hierarchy of nested subassemblies, and a procedure for integrating itself with its subassembly frames (a recursive process that resolves integration conflicts in favor of higher level subassemblies). The outputs are custom documents, typically compilable source modules.

A number of implementations of FT exist. XVCL is a general-purpose open-source implementation of FT. The Frame Processing Language (FPL) [FPL] is an XML language based on the Bassett's Frame Technology. FPL is implemented in Java.

Our work is similar to [JAR08] in its use XML-based technologies for generating the target domain member, but there are some major differences:

a) XVCL is a language with specific commands to manipulate the source code, and requires an interpreter, compiler and other associated tools. In contrast, our work we only uses XML and XML-based technologies that are open-source.

b) In his work, the generated code is mostly the expansion of the reusable parts of the source code according to XVCL commands. In contrast, while placeholders in the domain contract are indeed expanded into ACL 'code', generation depends on the feature grammar, which is absent from Jarzabek's approach.

2.3.1.2 Generative Programming

The work of Czarnecki and Eisenecker [CZA00] is considered a 'classical' generative technique and is at the root of our work in terms of inspiration and terminology. It provides mechanisms to manage and specify requirements for reuse. It is however very C++-dependent (and almost impossible to debug). Also, it must be pointed out that there is very little traceability between the domain engineering artifacts (i.e., feature diagram, feature grammar) and the domain code manipulated by their generator. The work of Smaragdakis in [SMA00] and [SMA02] is quite similar but instead of relying on C++'s partial template specialization, it instead promotes the use of parameterized inheritance, yet another language-specific feature [BAT92], [BAT03].

Furthermore, in [CZA05-b] and [CZA05-d], the authors introduced cardinality-based feature modeling (CBFM), in which they extended the original feature notation by FODA [KAN90]. They defined three different kinds of features as: root feature, grouped feature, and solitary feature. Along with that extension, they also introduced the concept of "staged configuration", more specifically in [CZA04-b] and [CZA05-a]. This concept can be obtained by stepwise specialization of a feature model with a multi-level configuration through a generative process.

A feature model represents all the possible configuration choices in a domain, where each configuration choice defines a set of specific features to be selected from the feature model, in order to generate the desired member of that domain. This process

can be done in multiple stages, where at each stage some of the configuration choices can be eliminated from the feature model, to create a new feature model at the next stage, hence called "staged configuration". At each stage a part of the desired member is generated, according to the choices available at that stage [CZA05-a]. They implemented a tool to apply their "staged configuration" and specialization on their CBFM model, as an Eclipse feature-plugin tool [ANT04].

Their specialization process is important in a realistic development process, as the configuration choices can be different from different domain requirements at different stages of the generative process, but the main point that remains in their work is that the level of features cannot be changed even through different specializations as they are always considered as abstract entities, while we need to generate artifacts closer to the concrete IUT items during our framework.

2.3.1.3 Requirements Library

There have been some attempts to semantically unify feature model notations (e.g., [SCH06], [CZA05-a], [CZA05-b], and [CZA05-c]). In [TAV07] we find a method to enhance expressiveness and use of a feature model that can support requirements reuse ([GRISS00], [JAC97]) in complex variability situations. These authors proposed a requirement library based on a new variability model in order to manage the common and variable requirements of the automotive domain at the requirements level. Their approach tackles a hierarchical domain with complex variability and even what they call 'variability of variability' (that is, variability is involved in generating of another domain). On major drawback however is that their solution does not address commonalities. Furthermore, it appears their approach is purely conceptual (that is, not implemented).

2.3.1.4 RMTs

In [HEI00-a], the authors proposed an extension to FODA [KAN90] in order to have domain knowledge include textual definitions of domain requirements. In that model, each requirement definition can address variability through parameters, thus each

parameter corresponds to a feature. This requirement model is viewed as a "requirement template". A requirement derivation process was proposed that instantiates the requirement template by replacing parameters with different variants, (i.e., feature values) in order to address a specific domain member. The authors described several problems they experienced using different requirement management tools and then made recommendations for such tools. They hinted at a partially automated derivation process.

In another paper by these authors [SCHL00], they claimed that many problems with respect to deriving requirements from the domain models can be handled by using knowledge-based systems, such as Konwerk [Ibid.]. Their results are based on a real industrial experiment [HEI00-b] on an industrial Car Periphery Supervision system.

2.3.1.5 DOPLER

In [RAB07] the authors presented a tool-supported product line engineering approach: Decision-Oriented Product Line Engineering for Effective Reuse (DOPLER). This tool consists of three integrated approaches based on the Eclipse platform:

- DecisionKing; to create variability models based on domain-specific meta-models, in which decisions can be defined for deriving products (that is domain members)

- ProjectKing; to plan the derivation and configuration process through the variability models

- ConfigurationWizard; to make decisions within different phases of the derivation process, to allow product customization, to capture requirements in order to generate a configuration

Decisions are written in natural language and use yes or no answers. The decisions can then be documented in a variability model, in which the dependencies among the decisions are represented via a graph or a tree structure. Also, during this process,

newly captured requirements not yet covered by the product line can be described using textual templates. At this point, the product configuration can be generated from the variability model using the Configuration Wizard tool. The generated configuration contains the selected features in tabular form, along with the templates of the new requirements in textual format. The authors claimed that the generated configuration can also be used to launch simulator applications based on the list of the selected assets for the final product. Relying on natural language makes this tool quite distant with respect to the problem we are addressing: one purpose of using ACL for requirements capture is precisely to avoid specifications in natural language, which are quite problematic for example with respect to their testability.

2.3.1.6 RED-PL

In [DJE07-a], [DJE07-b], and [DJE07-c], the authors proposed the RED-PL (Requirements Elicitation and Derivation for Product Lines) method that intends to support requirements derivation from a product line requirements' model (which constitutes the domain model). User product requirements are matched with the domain model. Specific requirements proceed from this matching process. RED-PL consists of three steps:

- Eliciting a stakeholder's requirements: The customers can specify their own requirements in their own way (!)

- Matching a stakeholder's requirements: a subset of the requirements captured in the domain model is identified as matching the needs of the stakeholder. An alternative set of requirements is also identified.

- Deriving a set of optimal requirements for the target product: A set of customer constraints (e.g., cost, development time and risk) as well as requirement alternatives are considered at this step.

It should be emphasized that there is no notion of a configuration defining a target member in this approach. Furthermore, Integer Linear Programming is used instead of generation proper. This requires expressing requirements and constraints using mathematical formulas. In discussing the first case study they carried out [DJE07-a], they report that such formalization was problematic for uses and that ultimately the matching step they envisioned did not work, nor was the approach found to be scalable. A better result was achieved in a second case study [DJE07-b] using a different constraint solver. But ultimately, there is no genuine generation in this approach despite its terminology. Their work proceeded a comparison among requirements variability modeling notations for product lines [DJE06].

2.3.1.7 XML-based Application Generators

Application generators were among the first generators developed. Application generators allow one to customize and reuse a specific set of software designs [CLEA01]. The input to an application generator is the specification of a problem or a task to be done by a program, and the output is one or more software products typically a segment of code, a subroutine, or a software system. In one paper [CLEA88] described the steps of building an application generator: first specify the requirements of an application generator for a specific problem domain. Then, based on these specifications and the target implementations, the generator was built using a set of transformation commands and grammar rules. The grammar rules define the rules for the specification language, and are saved as the source-description. The commands define templates for the output implementations and are saved as product-description. A product description also contains text that will be put directly into the output implementations. The template commands in a product-description document described how to take data, defined by grammar rules, and put it in the output implementations. Consequently, each new/modified implementation merely required to create/change the product-description document. This approach was based on Stage [CLEA88], a powerful application-generator that was developed at AT&T Bell Labs. Cleaveland concluded: "Application generators are difficult to build. They require

carefully designed specification languages, user interfaces, an intimate knowledge of the application domain, and the ability to design generic units of reliable software in the domain." As previously mentioned, we adopt an XML-based solution and thus such application generators are relevant.

2.3.1.8 PLUS

In [GOM04-a], Gomaa introduced a process called Product Line UML-Based Software Engineering Environment (PLUSEE) to support multiple view model of a software product line, integrating several tools for selecting features and checking feature dependencies. Then in [GOM05], he introduced Product Line UML-Based Software Engineering (PLUS) as a set of techniques that extend the UML methods for single systems (such as RUP [RUP]) to handle the product line systems. In previous papers [GOM04-b], and [GOM04-c] and then in subsequent papers [GOM06-a], [GOM06-b], and [GOM07], he suggested how UML can support multiple views at every phase of the generation of a product. For example, for domain analysis, Gomaa proposes both the class and the feature model views and explains the relationship between them via UML diagrams and annotations.

His proposal and its examples emphasize how such representations need to be close to concrete implementations (which is a major drawback).

2.3.1.9 Jezequel et al.

In [ZIA06], jezequel et al. introduced a model-driven framework based on UML models, in which they used *abstract factory* design pattern [GAMM95] to model design decisions. In their approach, which was initiated in [ZIA03-a], [ZIIA03-b], and [ZIA04], variability is modeled using UML 2.0 profile extended with tagged values and attached to some stereotypes. In general, their approach can be applied on both structural and behavioral views of an SPL model. In the former case, they start with a product line model based on class digrams along with decision models, and then derive a specific product model from it using INRIA Model Transformation Language (MTL). In the latter

case, the derivation process is applied on sequence diagrams to derive statecharts for a specific product model using a tool called Product Line Behavior Synthesis (PLiBS) [ZIA07]. In both cases pre- and post- constraints, implemented in OCL, should be met by the derived product model.

In their approach, although they define constraints for consistency checking for models, but they didn't provide any mechanism to implmenet this checking. They also stated that PLiBS does not support this checking either.

2.3.1.10 OSEK/VDX

OSEK/VDX [OSEK01] is an open architecture with specifications for an embedded operating system, communication capabilities and a network management component, which supports the reusability and portability of software applications. It uses a configuration tool and an automated generator in an embedded environment. More precisely, it takes a file created by its configuration tool and then generates application specific code based on that file. Most importantly, contrary to our approach, the set of possible configuration is hardwired.

2.3.1.11 Others

There are currently 103 model transformations scenarios in the large ATL library of transformations found at [ATL], and almost all of them use UML models. Also many other xml-based [ROH05] code generators exist that can be found at [XML-c]. Beyond these, there is still a multitude of code generators, which are not directly relevant to our problem. We briefly mention some in the rest of this subsection.

[EJBX] is an xml-based code generator developed by Houston Technology Group [HTG]. It takes xml-based definition input and generates code to support a list of EJB database management systems. More specifically it generates a database access layer to interface the back-ends of database management systems. It is quite foreign to our problem.

The Design Maintenance System (DMS) [DMS] is a software synthesis and re-engineering toolkit that enables the analysis, translation, and reverse engineering of large-scale "software" systems. The term "software" for DMS is very broad and covers any formal notation, including programming languages, markup languages, hardware description languages, design notations, data descriptions, etc. DMS can also be used for domain-specific program generation. A simple example of DMS could be an extremely generalized compiler that generates source code rather than binary code. Again, such generation is quite distant from domain engineering.

CodeWorker [CODW] is an open-source parsing tool and source code generator devoted to generative programming. It first parses the input code – in different languages – and then generates the desired output code. It provides various methods for parsing and code generation. It also provides a scripting language adapted both to the description of language grammars and to the writing of code generation templates. It is specifically for code generation and does not tackle feature modeling, feature grammars and domain modeling.

Jostraca [JOST] is a general purpose code generation toolkit. It accepts input in JSP syntax [JSP], combined with any other programming language syntax. The desired output code is embedded in the input using template scripts. It extracts the desired output by instantiating these templates.

SourceCafe [SOUR] is a Java code generator that automatically generates up to 80% of a web application code. It aims to be a rapid solution for database-driven web application development by generating a working prototype in a very short time (claimed less than a minute).

Quick [QUIC] is a Java-based package for working with XML files. It generates Java programs to process xml files. It uses a binding schema to map XML tags to Java classes.

JavaML [JAVA-b] is an xml-based representation of Java programs. It generates xml application that provides an alternative representation of Java source code.

Finally, in order to illustrate how widespread the notion of generation is, we remark that the Unix operating system [UNIX] offers a wealth of tools that can be used for building generators. Among them are Lex and Yacc, which are the most well-known code generators to produce lexical and syntactical analyzers – lexers and parsers. But, clearly, such tools are of little relevance for domain engineering.

2.3.2 Summary Table

The results of our survey are summarized in Table 2.2 below.

Table 2.2 - Summary of the Survey on Generative Approaches

Generative Approach & References	Domain Modeling Technique	Generative Technique Tx Prog	Technology & Tools	Dependency or Limitations	Summary of Approach
XVCL [JAR07] [JAR08] [XVCL], [FT], [FPL], [BAS97], [BAS07][XML-a] [XML-b][XML-c]	-Meta-level representation using XVCL	X	XVCL, XML, Frame, Java, JDOM-Saxon, Jakarta,	-Expansion of all combinations of reusable items, no configuration or selective commands -Software domain-specific	-Domain independent technique -Open-source technique -Roots in Frame technology -Extends reuse at architectural-level and structural-level -Using separation of concerns and XML-based variant configuration language (XVCL) tools for generating applications
Generative Programming [CZA00][CZA02] [CZA03] [CZA04-a] [CZA04-b] [CZA05-a] [CZA05-b] [CZA05-d] [CZA05-e] [ANT04][KAN90]	-Feature model [FODA] -Template-based specification in C++	X	Feature modeling tool, C++ Templates, C++ Compiler,	-Highly dependent to specification language due to its template construct -Variability is hard-coded in the specification -Specification lang. should have constructs to express the domain	-Specify requirements: variabilities and commonalities using a template-based programming language -A configuration list is made to address the desired member's requirements -Apply configuration list to template-based specification to generate the desired member -The generation process is done at the compile time
Requirements library [TAV07], [SCH06][KAN90] [CZA05-a], [CZA05-b], [CZA05-c], [DOORS]	-Feature model [FODA] -Requirement specification	_ _	dxl script, DOORS,	-No focus on and no reuse of commonalities -No details of specification and derivation methods -No implementation details of the tool	-Feature model is extended -Variability model is used to select the requirements and not for configuring them -Each requirements specification is a requirements library -Deriving specific requirements from the library by decreasing variabilities -Supporting tools are developed using Doors eXtension Language:dxl
[HEI00-a], [HEI00-b], [SCHL00], [DOORS] [KAN90]	-Feature model [FODA] -Textual requirement specification	_ _	RMT tools, DOORS,	-Requirements should be added to RMTs to support the partially automated derivation process	-Feature model is extended -Domain requirements are defined in textual format with variabilities through parameters -Requirement derivation is done by instantiating parameters with variants -Supporting tools include Requirement Management Tools (RMTs) like DOORS

Table 2.2 - Summary of the Survey on Generative Approaches (cont.)

Generative Approach & References	Domain Modeling Technique	Generative Technique Tx Prog	Technology & Tools	Dependency or Limitations	Summary of Approach
DOPLER [RAB07]	-Textual domain-specific meta-model & graphs	_ _	DOPLER, Eclipse, Configuration wizard,	-No focus on and reuse of commonalities -No details of specification and derivation methods -No implementation details of the tool	-Create decisions to customize a domain member in natural language -Variability model is created as a textual domain-specific meta-model -New requirements are described using Volere in textual format -Derivation and configuration of a domain member is done through the variability model -Derivation and configuration are supported by the automated tools
RED-PL [DJE07-a], [DJE07-b], [DJE07-c], [DJE06] [UCM] [KAN90]	-Feature model [FODA] - UML - UCMs - Aspects - goals	_ _	ILP, Excel, GNU-Prolog, UML tools, UML diagrams	-No configuration to select a member -Lack of ILP syntax -ILP scalability problem -Different levels of abstract for domain and user's requirements - No derivation and implementation details	-Domain requirements are modeled using feature diagram, UML diagrams, UCMs, aspects, goals -Users' requirements are matched with the domain's requirements -Using Integer Linear Programming (ILP) to define the constraints -Using mathematic formulas to develop the ILP models -Using Excel to analyze the ILP model -Supporting tool using GNU-Prolog is developed
Application generator [CLEA88], [CLEA01], [XML-a][XML-b] [XML-c]	-Requirement specification & template commands using XML	X	XML, XML tools, XSLT, XSLT tools, Stage, Java, Java Server Pages, Java Tag Libraries,	-Difficult to build -Need User-Interface -Need intermediate transformations	-Focus on separation of concerns: separating variability from commonality -A set of grammar rules are defined to specify requirements -A set of template commands are defined to generate the output code -Requirements are specified in source-description file -Template commands & output text are saved in product-description file -Applying template commands on requirement specification to generate the output code -For any new output code, changes are made in product-description file
PLUS [GOM04-a], [GOM04-b], [GOM04-c], [GOM05] [GOM07] [GOM06-a], [GOM06-b] [RUP][KAN90]	-Feature model [FODA] -UML-based meta-model	X	UML tools, UML diagrams	-UML meta-models are close to concrete implementation items	-Provides a set of techniques to extend UML methods to handle product line systems (PLs) -Domain requirements are modeled using feature diagram and UML diagrams with annotations -Using UML diagrams to represent multiple views of product generation at each phase -Using transformations among UML models to generate a product
Jezequel et al. [ZIA03] [ZIA04] [ZIA06] [ZIA07] [ECL]	-UML-based -OCL -Design pattern	X	UML tools -MTL -PLiBS -Eclipse	-UML meta-models -OCL meta-model	-Product line model s defined using UML class diagrams for static view & Sequence diagrams for behavioural view models -OCL constraints are imposed into model -Multiple transformtions for deriving product models
OSEK/VDX [OSEK01]	_	_ _	_	-Hardwired configuration space -Application specific	-Embedded environment with open architecture for generating applications -Takes configuration file (OIL) and generates application specific code

3 Phase I: Domain Engineering

In the next four chapters, we present an overview of the framework we have developed for modeling variability a domain using ACL contracts and obtaining the specific contract for a member of this domain using this domain model and a configuration list (or simply config-list hereafter) denoting this specific member. To do so, we will follow the development of our first case study throughout these chapters. For simplicity and clarity, we will focus on one specific subdomain of the general domain of sequential containers, namely that of List-based containers. The reason is simple: literature already exists [CZA00] for feature modeling of this subdomain, largely inspired by the work of Stepanov [STL95] on the implementation of such containers in the Standard Template Library [MUSS96] of C++.

Our proposed solution, summarized in Figure 1.3, consists of several *phases* [COR11-b]. Figure 3.1 expands Figure 1.3 with respect to the process of 'reducing' (or equivalently resolving) variability in order to produce a member's contract out of domain contracts (this terminology being explained when it becomes relevant).

The first phase of our approach addresses domain engineering [CZA00] and consists of 2 main steps, each discussed in a section of this chapter. These steps are performed manually, their deeply analytical nature making them difficult to automate.

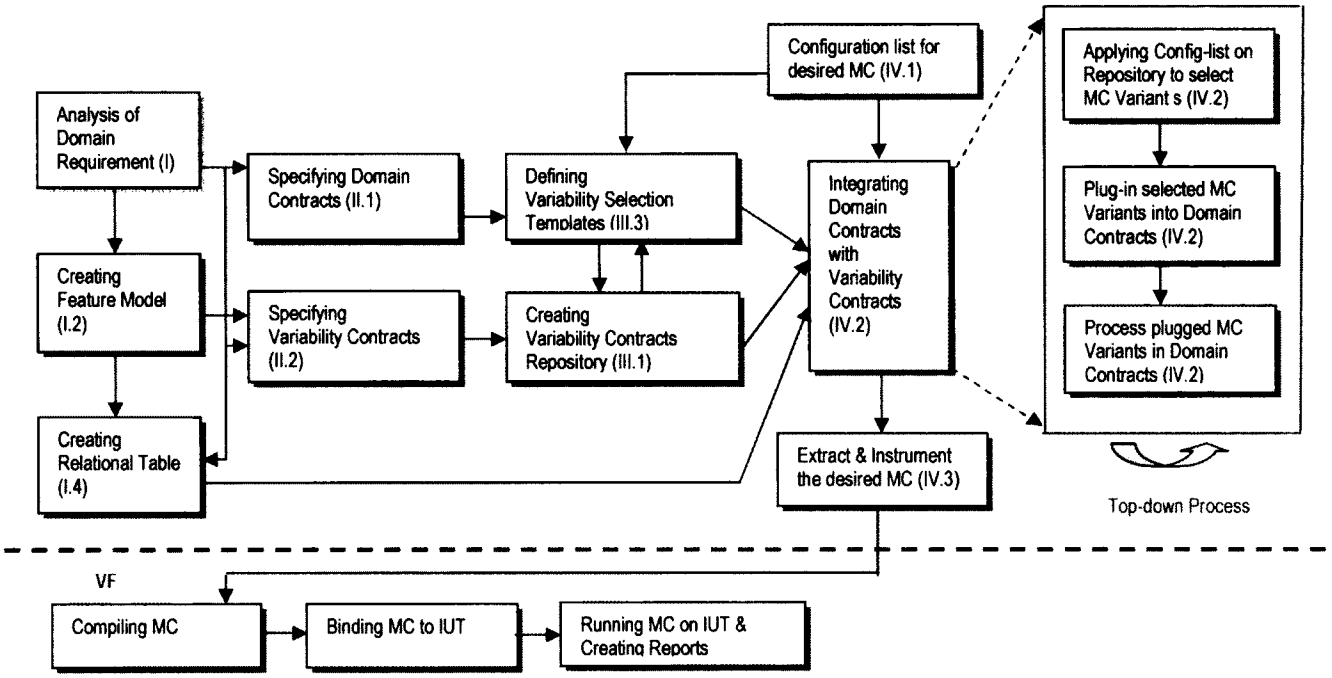

Figure 3.1 - Model-Driven Framework for Generating a desired Member Contract (MC) from a Domain Contract

3.1 From Domain Analysis to Domain Contract Elements

In step I.1, the domain engineer identifies commonalities and variability in the domain at hand. This analysis of the domain leads to the production of a feature model (step I.2). Figure 3.2 presents a portion of the feature model we start with for understanding commonalities and variability in the implementation of a family of list-based containers. The syntax of this model is that of FODA [KAN90] and similar notations [CZA00]. It is important to emphasize once more that our goal is to demonstrate the role of *a* feature model in capturing domain contracts. We will simply add that the illustrated model largely proceeds from augmenting the model proposed by Czarnecki and Eisenecker [CZA00] for list-based with specific considerations stemming from Stepanov's work [STL95] on STL.

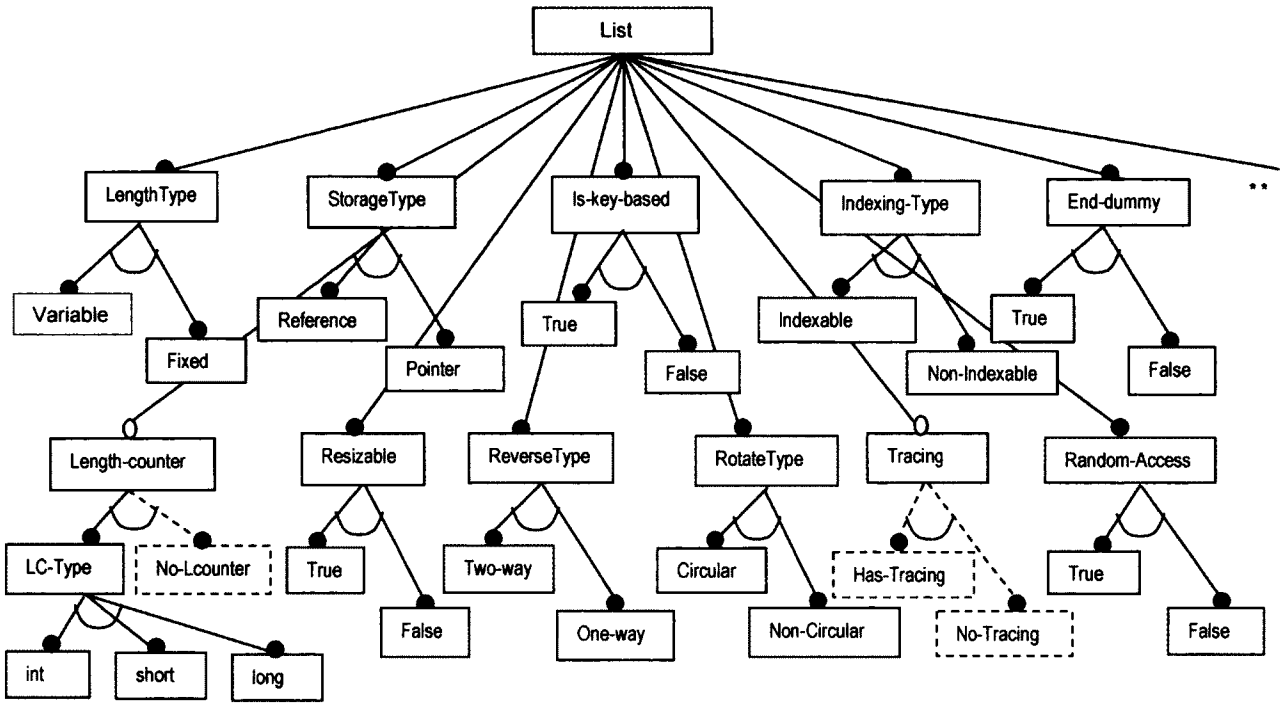

Figure 3.2 - Feature Diagram for Sequential Containers (partial)

At the end of step I-2, variation points and their variants have been identified and captured in a feature model. In this example walkthrough, we will restrict ourselves to the following self-explanatory variation points:

- Variation point VP0, named "Is-key-based", has two variants VP0-1 "True" and VP0-2 "False". It divides the general domain of containers into two parts: the subdomain of *sequential* containers (which do not use keys), and the domain of *associative* containers (which, conversely, rely on keys).

- Variation point VP1, "Length Type", has two variants VP1-1 "Variable" and VP1-2 "Fixed". It captures the fact that the containers, be they sequential or associative, can have a variable length or fixed length, where length is number of elements.

- Variation point VP2, "Storage Type", has two variants VP2-1 "Copy" and VP2-2 "Reference". It addresses whether a container keeps a copy of an actual item or a reference to it. This consideration may seem language dependent but, in fact, it is not (though the realization of each of these variants is): it is a fundamental design decision to know whether a container is to make copies or not of what it is given to store.

- Variation point VP3, "Resizable", has two variants VP3-1 "True" and VP3-2 "False". It indicates whether or not a container automatically resizes when full. How resizing occurs is implementation specific and of no importance here.

- Variation point VP4, "Rotate Type", has two variants VP4-1 "Circular" and VP4-2 "Non Circular". It models whether the last element links to the first one or not.

- Variation point VP5, "Reverse Type", has two variants VP5-1 "Two-way" and VP5-2 "One-way". A "two-way" sequential container allows going from one element to the previous one. In STL terminology, such a container supports a bi-directional iterator. By definition a sequential container always allows going from one element to the next, that is, supports at least a unidirectional iterator.

- Variation point VP6, "End-dummy", has two variants VP6-1 "True" and VP6-2 "False". If true, the container has a dummy node[3] at the end of itself. Meyer ([MEY92], [MEY94] and [MEY97) explains at length the advantages and disadvantages of such a design decision on the writing of contracts for containers. STL sequential containers use such a dummy end node.

- Variation point VP7, "Random-Access", has two variants VP7-1 "True" and VP7-2 "False". If true, elements can be accessed randomly, that is, by directly indexing to each element, as opposed to using linking sequentially from one element to the next. While sequential containers can offer random access, list-based ones typically do not.

[3] A dummy end node is an extra node at the end of a sequential container with the same structure as the other nodes but without the value content. It is considered in the design of STL sequential containers for multiple purposes, such as identifying the end of the container without traversing it from the begining.

- Variation point VP8, "Indexing-Type", has two variants VP8-1 "Indexable" and VP8-2 "Non-Indexable". This feature pertains to the underlying impementation of the container. If indexable, the underlying implementation of that container uses an array, otherwise a linked list (which is the subdomain we will mostly consider here). It should be noted that a sequential container can be indexable but disallow random access. For example a de-queue implemented using an array is indexable, but its elements cannot be accessed randomly due to its design.

- Variation point VP30, "Length-counter", is an optional feature (hence the use of dashed boxes) with one variant VP31 "LC-Type" [CZA99]. It captures whether or not a container keeps its current number of elements in a variable. What is interesting is that its variant (or feature value) is itself a variation point (i.e., a feature), as is often the case in feature models (see [CZA00]). Also, in our framework, capturing a variation point entails variant selection. In other words, if one introduces a feature, one must give this feature at least two values to choose from (which is not required in FODA [KAN90]). Thus, we require that VP30 have a second variant, variant VP30-2 "No-Lcounter" to capture the design decision of not having a counter for the elements of the container. We call such a variant, a virtual one, and denote it with a dashed box in our feature diagram. It is specific to optional features.

- Variation point VP31 "LC-Type" has three variants: VP31-1 "int", VP31-2 "short" and VP31-3 "long".

- Variation point VP33 "Tracing" which is an optional feature with no variants per se (as per [CZA00]) if captured in FODA. However, given how our feature selection mechanism works, we explicitly model a virtual variant "Has-Tracing" and another virtual variant "No-Tracing" (following Rules 1 and 2 of our feature modeling approach discussed at length in section 3.2)

This set of feature will suffice to illustrate the detail workings of our framework in the rest of this chapter.

The next step of phase I, step I.3, consists in identifying what are called the *reusable assets*, that is, the structural and behavioral components common to the domain.

With respect to structure, in ACL, any data inherent to a domain is to be captured using contract variables (see section 4.2) in the most abstract contracts associated with this domain. The domain of containers illustrates however a risk inherent to domain engineering, namely overgeneralization. For example, it is a mistake to introduce a contract variable at the highest level of a contract hierarchy for containers to capture the keeping of the current number of elements in some counter: the presence of VP30 and specifically of variant "No-Lcounter" in our feature model clearly indicates not all containers will possess such a counter. Using instead an observability at the level of the domain avoids this mistake (since the observability will be bound to a procedure that does or does not rely on a variable holding the current number of elements in a container, depending on the actual container at hand).

With respect to behavior, for containers, concepts inherent to the domain such as *allocation*, *insertion*, *deletion*, *traversal*, *search*, and *deallocation* should lead the domain engineer to recognizing corresponding responsibilities and scenarios, which are to be organized into a hierarchy of ACL contracts. For example, the *insertion* responsibility (see section 4.1) is a common behavioral requirement relevant to all the members of the domain. It is important to understand that while this ACL responsibility is common to the domain, its realization may differ from one member to another. For example, since a single-linked-list and a double-linked-list have different structures, they need to handle insertion (in terms of pre- and post-conditions) in different ways. Furthermore, the general idea of insertion may be refined in different ways: insertion at the front, insertion at the back, insertion at a given index (if indexable), etc. In I.3, we merely identify that insertion is a responsibility that is relevant to the whole domain and can be refined. Similarly, returning the size of a container (i.e., its current number of elements) is relevant to the whole domain. In this case however, given the semantics of ACL, the domain engineer must conclude that the domain requires a *size* Observability,

rather than a responsibility, the difference lying in the fact that, in ACL, an observability is like a read-only responsibility (see section 4.1).

Going from reusable ACL assets to domain contracts is not as simple as it may seem as it must take into account feature interactions, which we discuss next.

3.2 Building a Variability Relationships Table

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 [CZA00]. Consequently, it is often the case that the processing of such a feature grammar is entirely manual (e.g., [TAW12]). In contrast, in our framework, the feature interactions identified by the domain engineer are captured in a table whose use is later automated. We call such a table a feature grammar table, or equivalently a feature relational table (RT). The one for our example is given in Table 3.1. (The grammar for this table is given in Appendix A1.)

Table 3.1 - Variability Relationships Table (RT) for Sequential Containers, List members

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
2	VP1 VP1-1 VP1-2	Var. point Variant Variant	"LengthType" "Variable" "Fixed"	Rule 1, 2: VP1 ←X VP1-1 VP1 ←X VP1-2	VP1 <depends>VP1-1, VP1-2 Cond = - Action = variant
3	VP2 VP2-1 VP2-2	Var. point Variant Variant	"StorageType" "Copy" "Reference"	Rule 1, 2: VP2 ←X VP2-1 VP2 ←X VP2-2	VP2 <depends>VP2-1, VP2-2 Cond = - Action = variant
4	VP4 VP4-1 VP4-2	Var. point Variant Variant	"RotateType" "Circular" "Non-Circular"	Rule 1, 2: VP4 ←X VP4-1 VP4 ←X VP4-2	VP4 <depends> VP4-1, VP4-2 Cond = - Action = variant
5	VP5 VP5-1 VP5-2	Var. point Variant Variant	"ReverseType "Two-way" "One-way"	Rule 1, 2: VP5 ←X VP5-1 VP5 ←X VP5-2	VP5 <depends>VP5-1, VP5-2 Cond = - Action = variant
6	VP3 VP3-1 VP3-2	Var. point Variant Variant	"Resizable" "True" "False"	Rule 1, 2: VP3 ←X VP3-1 VP3 ←X VP3-2	VP3 <depends>VP3-1, VP3-2 Cond = - Action = variant
7	VP6 VP6-1 VP6-2	Var. point Variant Variant	"End-dummy" "True" "False"	Rule 1, 2: VP6 ←X VP6-1 VP6 ←X VP6-2	VP <depends>VP6-1, VP6-2 Cond = - Action = variant
8	VP7 VP7-1 VP7-2	Var. point Variant Variant	"Rand.-Acc." "True" "False"	Rule 1, 2: VP7 ←X VP7-1 VP7 ←X VP7-2	VP7 <depends>VP7-1, VP7-2 Cond = - Action = variant
9	VP8 VP8-1 VP8-2	Var. point Variant Variant	"Indexing-Type "Indexable" "Non-Indexable"	Rule 1, 2: VP8 ←X VP8-1 VP8 ←X VP8-2	VP8 <depends>VP8-1, VP8-2 Cond = - Action = variant
10	VP30 VP31 VP30-2	Var. point Var. point Variant	"Length-counter" "LC-Type" "No-Lcounter"	Rule 1, 2: VP30?←XVP31 VP30?←XVP30-2	VP30?<depends> VP31, VP30-2 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
12	VP33 VP33-1 VP33-2	Var. point Variant Variant	"Tracing" "On" "Off"	Rule 1, 2: VP33?←XVP33-1 VP33?←XVP33-2	VP33?<depends>VP33-1,VP33-2 Cond = - Action = variant
13	VP31 VP31-1 VP31-2 VP31-3 VP30	Var. point Variant Variant Variant Var. point	–	Rule 7: If(VP31 !=null && VP31 ==VP31-1 || VP31 ==VP31-2 || VP31 ==VP31-3) IMPLIES VP30?==LC-Type	VP31 <requires>VP30? = VP31 Cond1 = (VP30? == LC-Type) Action = variant1

Table 3.1 - Variability Relationships Table (RT) for Sequential Containers, List members (cont.)

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
14	VP3 VP3-1 VP1 VP1-2	Var. point Variant Var. point Variant	_	Rule 5: If (VP3==VP3-1) IMPLIES VP1 == VP1-2	VP3-1<requires>VP1-2 Cond1 =("VP1 == Fixed") Action= variant1
15	VP8 VP8-1 VP1 VP1-2	Var. point Variant Var. point Variant	_	Rule 5: If (VP8==VP8-1) IMPLIES... VP1==VP1-2	VP8-1<requires>VP1-2 Cond1 =("VP1 == Fixed") Action= variant1
16	VP7 VP7-1 VP8 VP8-1	Var. point Variant Var. point Variant	_	Rule 5: If (VP7 ==VP7-1) IMPLIES... VP8 == VP8-1	VP7-1<requires>VP8-1 Cond1 =("VP8 == Indexable") Action= variant1
17	VP30 VP1	Var. point Var. point	_	Rule 9: If (VP30? != null) IMPLIES... VP1 != null	VP30?<requires>VP1 Cond1 = ("VP1 == Fixed" || "VP1 == Variable") 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
20	VP4 VP4-2 VP5 VP5-1	Var. point Variant Var. point Variant	_	Rule 5: If(VP4==VP4-2) IMPLIES VP5 == VP5-1	VP4-2<requires>VP5-1 Cond1="VP5==Twoway" Action= variant1
21	VP4 VP4-2 VP5 VP5-2	Var. point Variant Var. point Variant	_	Rule 5: If(VP4==VP4-2) IMPLIES VP5 == VP5-2	VP4-2<requires>VP5-2 Cond2="VP5==Oneway" Action= variant2

The semantics of this table are reasonably straightforward. Column 2 refers to the ids of the variation point(s) and variants involved in a relationship. Column 3 spells out the nature of each such id. (i.e., whether it is a variation point or variant). Column 4 gives the name of a variation point if this row is used to defined a feature of the feature diagram and its variants. Column 5 spells out the relationship, referring to the rule(s) (discussed shortly) being used to define this relationship. Finally, column 6 captures whether this relationship involves any constraint (or condition) and, if so, what action should be performed, that is, which variant should be selected if this condition is true.

The actual specification of such an action can in fact occur later in the process (in Phases II and III) as will be explained in those phases.

First, rows 1 through 12 merely define the exact relationship between a variation point and its variants. For example, row 1 states that VP0 has two mutually exclusive variants VP0-1 and VP0-2. It also states that if the config-list (defining a specific member of the domain, as explained in chapter 1) 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). Then rows 13 to 21 deal with feature interactions proper. Row 13, for example, states that if VP31 is assigned any one of its valid variants in the config-list, then VP30 must also appear in the config-list and must be set to variant LC-Type. Rows 14 through 17 capture similar implications (i.e., this variant of that feature in the config-list entails this variant for that other feature). Rows 18 through 21 illustrate how multiple valid combinations of variants are handled. The key point to be grasped is that all such valid combinations of features and variants must be explicitly captured in this feature grammar. In the rest of this subsection, we discuss how one builds such a table.

Building this table involves the following steps:

1. Identifying each variation point and its related variants and entering them in 2nd column of RT. The id for a variation point x is "VPx" where x is the number assigned to it and the id for its variants is "VPx-y" where y is the number assigned to its variant. The 3rd column shows the type and the 4th column shows the real names for the variation point and its variants. The 1st column shows the row number in RT.

2. Identifying a relationship and its relevant rule(s): This relationship may be between a variation point and its own variants or other variants, between a variation point and another variation point, or between variants (as the rules explained shortly). Enter the rule numbers in column 5. If the relationship can hold under a condition go to step 3 otherwise go to step 4.

3. Identifying the conditions under which the relations can hold or not. Add the conditions in the 5th and 6th columns of the table using the syntax of the table and the syntax that appears in variability contracts (explained later) respectively.

4. Identifying the actions that should be performed if the conditions hold or not. Also identifying the actions that need to be done even without any condition. Add the actions in the 5th and 6th columns using the appropriate format.

5. Defining any precedence within conditions of a relationship and capturing it using compound conditions. This step is made necessary by the method that we use to 'resolve' variability, as will be explained in Phase III.

6. Considering the possible side-effects of each variability relationship: Put simply, each time a domain engineer identifies a variability relationship, she must consider this relationship entails further relationships (which will require being added to this table via new rows or modifications to existing ones).

7. Repeating steps 2 to 6 until the domain engineer determines all variability relationships have been captured. .

The order of the rows in the RT table is not relevant: it is only for readability that we start by considering the relationships between variation points and their own (direct) variants first and then the other relationships. But it must be emphasized that the generative process rests on this table: the presence of a variation point or a variant in the config-list may lead to a condition, the condition leads to an action, and the action may lead to expectations for another set of variation points, their constraints, actions, and so on.

The construction of the variability relationship table rests on a specific set of rules, which constitutes the semantic foundation of this work. We now discuss these rules in details.

According to [POH05] (example 4-18): "The mandatory variability[4] dependency states that a variant is required for a variation point to which it is related. This does not imply that the variant has to be included in all applications of the software product line. A mandatory variant is only part of an application if the related variation point is part of it." In this research, we will use this definition for mandatory variability. For example, in the domain of sequential containers, "Rotate-Type" and "Reverse-Type" are two mandatory variation points. To generate a linked-list container both of these variation points must specify actual variants in the config-list for that member. But for a Vector, these two variation points are irrelevant. Consequently, in the config-list for a Vector, these two variation points (which must appear in the config-list by virtue of being mandatory) will be set to some "null" (or semantically equivalent "don'tcare") value.

Following existing work on variability (see chapter 2), we use three categories of relationships:

1) Relationship between a variation point (VP) and a variant (Var): Each variation point must be related to at least one variant, and each variant must be related to at least one variation point

2) Relationship between a variation point (VP) and another variation point (VP): Each variation point can be related to other variation points

3) Relationship between a variant (Var) and another variant (Var): Each variant can be related to other variants

Variability Rules

A variability relationship can be defined with or without conditions (keywords <requires> and <depends> respectively), but always with an action. The general syntax of a variability "constraint" and its subsequent "action" is as follows:

[4] The mandatory variability dependency indicates the required relationship between each variation point and its valid variants according to FODA [KAN90].

IF <constraint>

IMPLIES <action>

The definition of a variability relationship must obey one of the following rules

Relationship between a variation point and a variant:

Rule 1: Variation point VPi is related to at least one variant VPi-1, represented as:

VPi <depends> VPi-1

Syntax in RT: *VPi ← VPi-1*

Semantics: Variantion point VPi is related to variant VPi-1 as one of its variants. This relationship can be implemented through "and", "or", or "x-or" relations among variants of one variation point FODA [KAN90]. The "and" relation (represented with ←) means that all the variants are necessary, the "or" relation (represented with ←R) means that some-of the variants are necessary, and the "x-or" or alternative relation (represented with ←X) means that one of the variants is necessary to have value in the config-list, if VPi is considered to be in the list. No condition is necessary to hold this relationship.

Implementation: If variant VPi-1 is selected for variation point VPi, then the action defined in its variability contracts should be plugged-in to the domain contract.

Rule 2: Variant VPi-1 is related to at least one variation point VPi, represented as:

VPi <depends> VPi-1

Syntax in RT: *VPi ← VPi-1*

Semantics: Variant VPi-1 is related to variation point VPi as one of its variants, with the same relationship defined in Rule 1 above. Variation point VPi can be related to another variation point VPj as its parent, and VPj to other variation points in the same

way [CZA00]. This means that variant VPi-1 can be related to more than one variation points: e.g., VPi and VPj. No condition is necessary to hold this relationship.

Implementation: If variant VPi-1 is selected for variation point VPi, then the action defined in its variability contract should be plugged-in to the domain contract.

Note – "Optional" Features and "OR" features: If a variation point is optional and it has no variants or just one variant, in order to use it and find its relationships with the other variation points and variants in our work, we consider either one or two virtual variant(s) for that variation point. For example in the domain of sequential containers VP33 "Tracing" is an optional feature with no variants. For this variation point, we consider two virtual variants – represented with dashed lines in our feature diagram – variant VP33-1 "On" and variant VP33-2 "Off." We call these virtual variants.

Also, we will use the same method for the "OR"-ed features. If the variants are related to their variation point with an "OR" relation, this means that either any of them or a combination of them can be selected to instantiate their variation point. For the same reason as for the optional features, we consider extra virtual variants beside the "OR" variants, to analyze all the possible options.

Examples are given in the relationship table above.

Before considering further relationship rules, we need to clarify their interpretation through some general rules.

General Rules for Variability Relationships

Consider VPi has a variability relationship with VPj based on condition "cond1", then:

1- If during subsequent analysis, an additional condition is discovered for this relationship, it can be combined to "cond1" only if it is not redundant in "cond1".

2- If "cond1" applies to VPi regardless of its variants, then, conceptually, "cond1" is applied to each of the variants of VPi. (see Rules 3 and 9 below for more details).

3- Specifying condition (VPj != null), means checking if variation point VPj is equal to any of its variants as: "VPj==VPj-1 || VPj==VPj-2 || ..." (see Rule 3 and 9 in RT table for more details).

4- In case other conditions for VPi cannot be combined with "cond1" (because they are mutually exclusive), then these other conditions should be defined separately (e.g. in "cond2") with their relevant actions (e.g. in "variant2").

5- In case some variants are mutually exclusive, then a separate condition (e.g. "cond2") should be defined to introduce such special cases with their relevant actions (e.g. "variant2") (see example in Rule 6 below for more details).

6- This type of relationships between a variant and another variant is considered under Rule 5 below in our variability contracts. (see table above for examples).

Relationship between a variation point and a variant:

Rule 3: Variation point VPi requires variation point VPj with selected variant β, in order to function correctly, represented as:

VPi <requires> VPj = β

Syntax in RT: IF (VPi != null && VPi != dontcare) IMPLIES VPj == β

Semantics: if variability VPi is specified in a configuration list (i.e., it is neither null nor dontcare) then, regardless which variants of it is selection, variant β should be selected for VPj.

Implementation: This variability constraint, which is represented in RT as "VPj == β": 1) should be combined with the other conditions in the variability contracts for VPi using "and" operator making a compound condition named "cond1" (if allowed by the general

rules), 2) then "VPj == β" should be checked in the RT table to determine if it requires "condγ", and if yes "cond1" should be "anded" with "condγ", 3) then "VPi != null" should be checked in RT table to determine if any other variation point VPn requires this condition. If so the conditions for VPn should be "anded" with "cond1". If "cond1" is true, the action named "variant1" defined in the variability contracts for VPi should be plugged-in to the domain contract.

Note: the implementations of all remaining rules are similar to the implementation of Rule 3 and thus are not repeated.

Rule 4: Variation point VPi requires variation point VPj not to select variant β[5], in order to be excluded, represented as:

VPi = null <requires> VPj != β

Syntax in RT: IF (VPi == null || VPi == dontcare) IMPLIES VPj != β

Semantics: same as for Rule 3 except here variant β should not be selected for VPj.

Example: see variation point "Is-key-based" in chapter 7.

Relationship between a variant and another variant:

Rule 5: Variability VPi with specific variant α "requires" variability VPj with specific variant β to function properly, represented as:

VPi = α <requires> VPj = β

Syntax in RT: IF VPi == α IMPLIES VPj == β

Semantics: If variant α is selected for variability VPi, this implies that variant β should be selected for VPj. In other words if variant α is selected then variant β should be selected in the config-list too.

[5] Symbol β represents a valid variant of variation point VPj.

Rule 6: Variabilities VPi and VPj are mutually exclusive based on their specific variants, represented as:

VPi = α <requires> VPj != β

Syntax in RT: IF VPi == α IMPLIES VPj != β

Semantics and implementation are similar to rule 5.

Example: see variation point "Iterator-Type" and "Reverse-Type" in section 4.2: variant "Bi-direction" for "Iterator-Type" rules out variant "one-way" for "Reverse-Type".

Relationship between a variation point and a variant:

Rule 7: Variability VPi requires optional variability VPj with the selected variant α to function correctly, represented as:

VPi <requires> VPj? = α

Syntax in RT: IF (VPi != null && VPi != dontcare)

IMPLIES (VPj? != null && VPj? != dontcare && VPj? == α)

Semantics: If variability VPi is specified in configuration list (i.e., it is neither null nor dontcare) then regardless of its selected variants, this implies that optional variability VPj must be specified in the configuration list (i.e., it should be neither null nor dontcare).

Rule 8: Variability VPi requires optional variability VPj? = null or specific variant β[6], in order to be excluded, represented as:

VPi = null <requires> VPj? = β

[6] Symbol β represents a valid variant of optional variation point VPj.

Syntax in RT: IF (VPi == null || VPi == dontcare)

IMPLIES (Vpj? == null || VPj? == dontcare || VPj? == β)

Semantics: If optional VPj should not be in the configuration list (i.e., it is either null or dontcare) or if it selects variant β, this implies that non-optional variability VPi is not to be specified in the configuration list either, that is, it is either null or dontcare.

Example: In the domain of sequential containers, variation point "Length-counter" has one variant "LC-type" which is itself another variation point with three variants. As "Length-counter" is an optional feature with only one link to its child feature "LC-type", we consider another variant for it named "No-Length-counter".

1) To consider "LC-type" in the config-list (it is neither null nor dontcare) and regardless of any selection of its variants, it requires considering variation point "Length-counter" in the config-list to be equal to "LC-Type". So variation point "LC-type" requires its optional parent variation point, in order to work correctly (Rule 7).

2) If optional variation point "Length-counter" should not be considered in the config-list (it is either null or dontcare) or if it can be equal to its specific variant "No-Length-counter", this automatically excludes variation point "LC-type" from the config-list (Rule 8).

Relationship between a variation point and another variation point:

Rule 9: Variability VPi requires variability VPj to function correctly, represented as:

VPi <requires> VPj

Syntax in RT: IF (VPi != null && VPi != dontcare)

IMPLIES (VPj != null && VPj != dontcare)

Semantics: If variability VPi is specified in the configuration list, this implies that variability VPj is to be considered in the configuration list too (it should be neither null nor dontcare), regardless of any variants being selected for each of them. In other words these variation points require each other.

Rule 10: Variability VPi does not require variability VPj to function correctly and vice versa represented as:

VPi = null <requires> VPj

Syntax in RT: IF (VPi == null || VPi == dontcare)

IMPLIES (VPj != null && VPj != dontcare)

Semantics: If variability VPi should not be considered in configuration list (i.e., it is either null or dontcare), then variability VPj should be considered in the configuration list (i.e., it is neither null nor dontcare), regardless of any variants being selected for each of them, these variation points exclude each other.

Example: In case study II variation point "key-ordering" has two variants and variation point "key-compare" has two variants too. Any selection of variants for variation point "key-ordering" requires the selection of a variant for variation point "key-compare". So regardless of their variants, these two variation points require each other in the configuration list (Rule 9).

Phase I concludes with the creation of a traceability table. The purpose of this table is to link variation points between them and with any relevant variants (as per the relationship table), then in phases II and II with the relevant domain and variability contracts. Our approach to traceability is summarized in sub-section 6.2.1.

4 Phase II: Domain and Variability Contracts

In this phase contracts for variabilities and commonalities of the domain of sequential containers are specified based on the results from phase I.

4.1 Specifying Domain Contracts using ACL

According to contract hierarchy for sequential containers, the common structural and behavioral aspects of the domain – domain commonalities – are specified in the form of a hierarchy of contracts using ACL. These contracts, named domain contracts, include, at the root level, the abstract requirements of the domain of sequential containers. Those are captured in the abstract contract "ContainerBase", which follows:

```
abstract Contract ContainerBase<Type T>
   {
      // defining variables
      Scalar Integer v;
      Scalar Integer old_Size;
      Scalar Integer Size;

//observabilities without bodies will be bound to corresponding IUT methods
      Observability INT size();
      Observability INT max_size();
      Observability Boolean IsEmpty();
      Observability Boolean IsFull()
      {  //plugin VP1();
      }

      // new() will be bound to any type of constrcutors
      Responsibility new()
      { //allocate memory for container
         Size = 0;
         Post(IsEmpty()== true);
      }

      // finalize() will be bound to any type of destructors
      Responsibility finalize()
      { //free the allocated memory for container
         Pre(size() == 0);
         Pre(IsEmpty() == true);
      }

      Responsibility GenericInsertion()
      { //cannot insert into a full container
         Pre(IsFull() == false);
         old_Size = size();
         Execute();
         Size = Size + 1;
         Post(size() == old_Size + 1);
      }

      Responsibility GenericDeletion()
      {//cannot delete from empty container
         Pre(IsEmpty() == false);
         old_Size = size();
```

```
            Execute();
            Size = Size - 1;
            Post(size() == old_Size - 1);
        }

        Invariant SizeCheck
        {//check if the Size is equal to the IUT container size
            Check(context.Size >= 0);
            Check(context.Size == size());
        }

    }// End Contract
// ------------------------------------------------------------------------
```

Generic contracts specify a list of generic parameters, variables, and functionalities that are general purpose and can be reused across the domain for the more specialized contracts. "ContainerBase" is a generic contract. ContainerBase has a generic input parameter named T that can be the type of the container elements in the IUT. Most of it should be self-explanatory but we offer here a brief explanation of the relevant aspects of this contract:

After the input parameter, we have the declaration of some contract variables that will be used in the contract body and are independent from the IUT variables [ACL]. Variable old_size is used to store the number of items in the container before any insertion or deletion in it. Variable Size is used to store the number of items that are in the container.

Following these declarations of variables, we have the list of Observability methods, which are similar to read-only methods and procedures in actual IUT. An observability in a contract represents an observation requirements for the corresponding method in IUT. The observabilities without the definition will be bound to the corresponding methods in the IUT and will return the return-value from that method. The observabilities without the bodies will not be bound to any IUT methods and will only be used for the observation purposes inside the contract.

Observabilities size() and max_size() will be bound to the corresponding IUT methods and will return the current size and the maximum size of the container respectively in generic INT type, while the IUT is being executed. The observabilities

isFull() and isEmpty() will be bound to the corresponding methods in the IUT and will return Boolean true if the container is full or empty respectively, and will return Boolean false otherwise.

After the definition of observabilities, the definition of responsibilities (which are similar to methods and procedures in actual IUT) can start.

Responsibility GenericInsertion() will be used as a generic method for all the methods that insert or add an element to the container. The element to insert and the return type will be defined by each responsibility that will use and refine this responsibility. The statements that will be used by any add or insertion method for the sequential containers before the execution will be: A pre- condition to ensure that the container is not isFull( ), saving current size of the container. And after the execution will be: A post- condition to ensure that the container size has been incremented by 1. We also update the number of elements – in the actual IUT container – in the contract variable Size (this variable will be checked later).

Responsibility GenericDeletion() will be used as a generic method for all the methods that delete or remove an element from the container. Again the return type will be defined by each responsibility that will use and refine this responsibility. The statements that will be used by any delete or remove method for the sequential containers before the execution will be: A pre- condition to ensure that the container is not isEmpty( ), saving current size of the container. And after the execution will be: A post- condition to ensure that the container size has been decremented by 1. We also update the number of elements – in the actual IUT container – in the contract variable Size (this variable will be checked later).

In our base contract, the Invariant checks compare the size of the container (the actual container in IUT) with the Size in the contract at all the time of the IUT execution. They should be matched together, and if not, an error will be reported by the contract.

The context keyword is similar to the *this* keyword in C++ or Java languages, and it refers to current contract that it is defined in.

It should be noted that the abstract contract and its body – the observabilities and the responsibilities defined in abstract contract – are not supposed to be bound to any IUT counterparts, but rather they are defined to be refined and reused by the contracts at the next levels that will be bound to the IUT to be executed. We will see the latter contracts in the following.

More of the domain contracts are captured in the MainContract, which follows:

```
// ---------------------------------------------------------------------
MainContract Container extends ContainerBase<tItem>
    {
        //defining variables
        Scalar tIterator tobeNext;
        Scalar tIterator tobePrev;
        Scalar tIterator new_pos;
        Scalar tIterator pos;

        // The size(), max_size(), and IsEmpty() observabilities
        // will be copied to this contract
        Observability Integer Front();//returns value of the first node
        Observability Integer Back();//returns value of the last node
        Observability tIterator Begin();//returns pointer to first node
        Observability tIterator End();//returns pointer to last node
        Observability tItem itemAt(tIterator pos);//returns item at pos
        Observability tIterator findNext(tIterator pos);//point to next
        //plugin VP5();

        //indexing an item in an indexable container
        //plugin VP8();

        //tracing links for variability
        //plugin VP33();

        //returning pointer to previous node
        Observability tIterator find_prev(tIterator pos)
        {
            //plugin VP5();
        }

        //checking random access return in indexable container
        Observability Boolean check_itemAt(Integer n)
        {   //plugin VP7();
        }

        Observability Boolean in_range(Integer index)
        {
            value = (index>0 && index <= size());
        }

        Observability Boolean checkNext(tIterator pos, tIterator tobeNext)
```

```
{
    Boolean result=false;
    result = (findNext(pos) &= tobeNext);
    value = result;
}

Observability Boolean checkPrev(tIterator pos, tIterator tobePrev)
{
    Boolean result=false;
    result = (find_prev(pos) &= tobePrev);
    value = result;
}

refine Responsibility new()
{
    Post(End() == Begin());
    //plugin VP6();
}

refine Responsibility finalize()
{
    fire(ContainerDone);
}

// Inserting an item at the beginning of the container
Responsibility Void Push_front(tItem item) extends GenericInsertion()
{
    tobeNext = Begin();
    Execute();
    Post(itemAt(Begin()) == item);
    checkNext(Begin(), tobeNext);
    //plugin VP4();
}

// Inserting an item at the end of the container
Responsibility Void Push_back(tItem item) extends GenericInsertion()
{
    tobeNext = End();
    tobePrev = find_prev(End());
    Execute();
    new_pos = find_prev(End());
    Post(itemAt(new_pos) == item);
    checkNext(new_pos, tobeNext);
    checkPrev(new_pos, tobePrev);
    //plugin VP4();
}

// Inserting an item before a position
Responsibility tIterator Insert(tIterator pos, tItem item)
    extends GenericInsertion()
{   tobeNext = pos;
    tobePrev = find_prev(pos);
    Execute();
    Post(value not= null);  //return-value from IUT insertion
    Post(itemAt(value) == item);
    checkNext(value, tobeNext);
    checkPrev(value, tobePrev);
    //plugin VP4();
}

// Inserting an item after a position
Responsibility tIterator InsertAfter(tIterator pos, tItem item)
     extends GenericInsertion()
```

```
    {
        Pre(pos not= End());
        tobeNext = findNext(pos);
        tobePrev = pos;
        Execute();
        Post(value not= null);  //return-value from insertAfter
        Post(itemAt(value) == item);
        checkNext(value, tobeNext);
        checkPrev(value, tobePrev);
        //plugin VP4();
    }

    // Deleting an item from the beginning of the container
    Responsibility Void Pop_front() extends GenericDeletion()
    {
        pos = Begin();
        new_pos = findNext(Begin());
        Execute();
        Post(Begin() == new_pos);
        //plugin VP4();
        //plugin VP2();
        }

    // Deleting an item from the end of the container
    Responsibility Void Pop_back() extends GenericDeletion()
    {
        new_pos = find_prev(End());
        pos = new_pos;
        tobePrev = find_prev(new_pos);
        Execute();
        checkPrev(End(), tobePrev);
        //plugin VP4();
        //plugin VP2();
        }

    // Deleting an item from a position
    Responsibility Void Erase(tIterator pos) extends GenericDeletion()
    {
        Pre(pos not= End());
        tobeNext = findNext(pos);
        tobePrev = find_prev(pos);
        Execute();
        checkNext(tobePrev, tobeNext);
        checkPrev(tobeNext, tobePrev);
        //plugin VP4();
        //plugin VP2();
        }

    //Resizing the fixed size container
    Responsibility Void resize()
    {//plugin VP3();
    }

Exports
    {
        Type tItem conforms Item
        {
            not context;
            not derived context;
        }

            Type tIterator conforms Iterator;
```

```
            //plugin VP0();

            //plugin VP30();
            //Type INT conforms Integer;

            //plugin VP31();
    }
}// End Contract
```

This contract depends on two short contracts (Iterator and Item) that are not relevant to variability. This contract inherits from the abstract contract above and refines its content to produce specialized contracts for the linked-list sequential containers. The input parameter to this contract is the type "tItem" which is a generic type for each element's of the container. This generic type will be defined as the actual type of the IUT elements, at the end of this contract. Each container element is specified through contract Item. The contract Item defines that each item has a value part that should be set after the item is created through responsibility new( ), and before the item is destructed through responsibility finalize( ). The observability Value( ) will return the value part of the element which is a scalar type – not a list.

At the beginning of the contract we have the definition of contract variables and then the observabilities without bodies that will be bound to the corresponding IUT methods. These observabilities are defined according to the specification provided for the sequential containers in [STL95]. In this example, for simplicity, we will focus on the insertion and the deletion responsibilities among the linked-list sequential containers. As such, besides the value part, we need a second part at each item structure that keeps the pointer to the next element – for the single linked-lists – and even the third part that keeps the pointer to the previous element – for the double linked-lists.

Observabilities Begin() and End() are defined to return the iterators – pointers – to the first and last elements of the container. Also observabilities findNext() and findPrev() are defined to return the iterators – or pointers – to the next and the previous elements in the linked-list given the current element's position as the input parameter. Observabilities Front() and Back() return the value of the first and last elements of the

container. And observability itemAt() will return the whole item of the given element's position.

The above observabilities will be bound to the corresponding IUT methods in order to be executed, and for this reason they all have the same signatures as their relevant methods in IUT. The following observabilities don't need to be bound to IUT methods, as they are specified only to be used inside our contracts:

Observability checkNext() has two input iterators. It checks if the second iterator points to the next element of the element positioned in the first iterator. If yes, the output is Boolean true and false otherwise.

Observability checkPrev() has two input iterators. It checks if the second iterator points to the previous element of the element positioned in the first iterator. If yes, the output is Boolean true and false otherwise.

Next, the responsibilities new() and finalize() which were already specified in the abstract contract are refined now using the keyword refine. Using this keyword means to refine the abstract responsibility that have the same signature with the one with refine keyword. The refinement is achieved by adding the content of the responsibility with refine keyword to the content of abstract responsibility.

To the abstract responsibility new(), a post- condition will be added that checks if the iterators pointing to the first and the last elements of the container are the same or not. And if they are, this means that the linked-list doesn't have any elements. According to the specification provided in [STL95], the last element in a linked-list is always followed by an extra element which is pointed to by the End() method. Also at the time of creating a linked-list, the only element that should existed in the list is the one pointed to by End(). This dummy end element does not contain any value.

There is also a refinement for the responsibility finalize() to add a fire() statement to it. The fire() statement is used to generate the observable event named ContainerDone, and this event indicates that the task is completed (or the destructor is done in the IUT).

The following responsibilities will be bound to the corresponding IUT insertion and deletion methods and all of them are refined versions of the Generic responsibilities specified in the abstract contract, through the keyword "extends".

When refining with keyword "extends", the code placed before the Execute() statement in the Generic responsibility will be added at the beginning of each of the following insertion and deletion responsibilities, and the code placed after the Execute() statement in Generic responsibility will be added right after the Execute() statement in the following insertion and deletion responsibilities [ARN09-a].

The responsibility Push_front() inserts a new element at the beginning of the container – to be the new first element. This responsibility uses the content of the GenericInsertion through the keyword extends. It checks if the new inserted element contains the correct item equal to the input item, through a post- condition. It also checks if the next element's position to the new inserted position is the same as the previous Begin() position, through the observability checkNext(). For this reason the previous Begin() position was already saved in variable tobeNext.

The responsibility Push_back() inserts a new element at the end of the container – to be the new last element located before the End() element. This responsibility uses the content of the GenericInsertion through the keyword extends. It checks if the new inserted element contains the correct item as the input item, through a post- condition. It also checks if the previous element to the new insertion position is the previous last element through observability checkPrev(), and if the next element to the new insertion position is the previous End() element through observability checkNext(). It should be noted that before and after the insertion the End() element should be the same. For this

reason, before the insertion, the End() element position was saved in variable tobeNext, and the last element position was saved in variable tobePrev.

The responsibility Insert() inserts a new element before the input position in the container. This responsibility uses the content of the GenericInsertion through the keyword extends. After the IUT execution for the corresponding insert method, the return value from the IUT is saved in the contract variable value. The value is the position – iterator – of the new inserted element by the IUT, and so it is checked not to be null, means that the insertion was occurred properly. Also the item at value is checked to be the same as the input item. These checks are performed through post-conditions. It also checks if the positions of the previous and next elements to the new inserted element are correct, through the observabilities checkPrev() and checkNext() respectively. For this reason, before the insertion, the input position itself was saved in variable tobeNext, and the position of the previous element of it was saved in variable tobePrev.

The responsibility InsertAfter() inserts a new element after the input position in the container. This responsibility uses the content of the GenericInsertion through the keyword extends. After the IUT execution for the corresponding insert method, the return value from the IUT is saved in the contract variable value. The value is the position – iterator – of the new inserted element by the IUT, and so it is checked not to be null, means that the insertion was occurred properly. Also the item at value is checked to be the same as the input item. These checks are performed through post-conditions. Before the IUT execution, the input position was checked not to be equal to the End() position through a pre- condition, as no element can be saved after the End() position according to [STL95]. It also checks if the positions of the previous and next elements to the new inserted element are correct, through the observabilities checkPrev() and checkNext() respectively. For this reason, before the insertion, the input position itself was saved in variable tobePrev, and the position of the next element of it was saved in variable tobeNext.

The responsibility Pop_front() deletes an element from the beginning of the container – to make a new first element. This responsibility uses the content of the GenericDeletion through the keyword extends. It checks if the new Begin() element is at the correct position, through a post- condition. For this reason, before the deletion, the position of the next element to the Begin() was found through observability findNext() and was saved in variable new_pos.

The responsibility Pop_back() deletes an element from the end of the container – to make a new last element. This responsibility uses the content of the GenericDeletion through the keyword extends. It checks if the new last element is at the correct position, before the End() position, through the observability checkPrev(). For this reason, before the deletion, the position of the two prior elements to the End() was found through observability findprev() and was saved in variable tobePrev. As it was mentioned before, the last element in a linked-list container is the element previous to the End() element [STL95]. As such by deleting this last element, the element before it which is two elements prior to the End() will be the new last element.

The responsibility Erase() deletes an element from the exact input position from the container. This responsibility uses the content of the GenericDeletion through the keyword extends. It checks if the input position not to be equal to the End() position, because there is no actual element at End(). It also checks if the positions of the next and the previous elements to the deleted position are correct after the deletion, through the observabilities checkNext() and checkPrev() respectively. For this reason, before the deletion, the positions of the next and the previous elements to the input position were found through observabilities findNext() and findPrev, and they were saved in variable tobeNext, and tobePrev respectively.

Scenarios and metrics are not discussed here as currently they are not addressed by our approach to variability.

The contracts presented above not only capture commonalities: they have been augmented to mark where variability is relevant in them through the use of the new plugIn marker, as in: `//plugin VP4();`

These place holders are in fact only added in step III.2, as will be explained later. We included them here in order to avoiding repeating the whole contracts just to show them. In other words, the outcome of step II-1 is a set of domain contracts in ACL (i.e., a model of the domain commonalities) without any variability.

4.2 Specifying Variability Contracts

After specifying domain contracts in step II-1, we can now model variability across this domain. To do so consists in our work in specifying *variability contracts* for the domain using a new syntax we call ACL-V[7] (see Figure 1.4). A variability contract captures a variation point, its possible variants, their semantic relationships, and finally how each variant, if it is present in the config-list at hand, refines the domain contracts. There are two possibilities for such modifications to the domain contracts: refinement by addition (keyword Refine-a) or refinement by replacement (keyword Refine-r, see the future work in updating our generative process to include scenario variability).

For the domain of sequential containers, proceeding from conceptually integrating the relationship table with the domain contracts, we obtain the following variability contracts (which should be self-explanatory given the supplied comments):

```
Variability Contrcats: Sequential Containers
{
  Variation VP0 <Is-key-based> [1..1] out of 2
  /* if this is an associative container then in the contract at hand
     they type key_type must conform to the contract Key.
     Otherwise key-type is not modified by the selection of variant VP0-2.
     The Refine-a keyword specifies that the definition of type key_type
     is to be found in the Exports section of the contract at hand.
  */
  {
```

[7] ACL-V language is developed to support variability contracts in the contract-based specifications written in ACL. The grammar and syntax of ACL-V is separate from ACL and so it is not an extension to ACL. For more details about the grammar rules and syntax of ACL-V, please refer to Appendixes 1 and 2 of this document.

```
  case "True":
    plug-in: VP0-1 //Item is associated with a key
    Refine-a: Exports
    Type key_type
    {
          Type key_type conforms Key;
    }
  case "False":
    plug-in: VP0-2 //Item is not associated with a key
    Refine-a: Exports
    Type key_type { }
}

Variation VP1 <Length-Type> [1..1] outof 2
/* if variable length then Observability isFull necessarily returns false
   Otherwise, current size is checked against max_size
 */
{
  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());
    }
}

Variation VP2 <StorageType>[1..1] outof 2
/* A postcondition is added to three responsibilities IF the container
   keeps references.
   That is, if variant is "Reference" then each element keeps a pointer
   to the original value and therefore after deleting that element, its
   pointer must be null (ie not pointing to anything), which explains
   the precondition. For variant "Copy" no such post-condition is needed.
*/
{
 case "Copy":
    Plug-in: VP2-1 //container element binds to a value type
    Refine-a: Responsibility
    Void Pop_front(), Void Pop_back(), Void Erase(tIterator pos)
    {}
 case "Reference":
    Plug-in: VP2-2 //container element binds to a pointer type
    Refine-a: Responsibility
    Void Pop_front(), Void Pop_back(), Void Erase(tIterator pos)
    {
       Post(pos == null);
    }
}

Variation VP3 <Resizable>[1..1] outof 2
/* When a container is resizable (VP3-1) AND has fixed size (VP1)
    then responsibility resize is defined to double the size of the
    container if we are at max_size.
  If VP1 is not Fixed, COND1 will be false, in which case resize
    is not required.
```

```
  This variability contract corresponds to row 15 of the RT table.
*/
{
 case "True":
   Plug-in: VP3-1 //container has a fixed resizable length
   Cond1: ("VP1 == Fixed")
   Refine-a: Responsibility
   Void resize()
   {
     #IF (Cond1)
      Pre(size() == max_size());
      old_size = size();
      Execute();
      Post(max_size() == old_size * 2);
     #ENDIF
   }
 case "False":
   Plug-in: VP3-2 //container has a fixed non-resizable length
   Refine-a: Responsibility
   Void resize()
      {}
}

Variation VP4 <RotateType>[1..1] outof 2
/* This variability contract corresponds to rows 18 through 21 in the RT
table. It considers all the combinations of variants for VP4 and VP5.
This is made necessary because each such combination uses a distinct set of
pre- and post-conditions for a set of 7 responsibilities!
Semantically, these combinations of variants each lead to rules making
sure that if the container is circular, the last element and the first one
are properly linked, which depends on whether the container is one-way or
two-way.
This variability contract illustrates the general mechanism we have created
to deal with specific configurations of variants.
*/
{
 case "Circular":
   plug-in: VP4-1 //container has a circular structure
   Cond1: ("VP5 == Two-way")
   Cond2: ("VP5 == One-way")
   Refine-a: Responsibility
      Void Push_front(tItem item), Void Push_back(tItem item), tIterator
   Insert(tIterator pos, tItem item), tIterator InsertAfter(tIterator pos,
   tItem item), Void Pop_front(), Void Pop_back(), Void Erase(tIterator pos)
   {
     #IF (Cond1)
      {  tobeNext = Begin();
         Post(checkNext(End(), tobeNext) == true);
         tobePrev = End();
         Post(checkPrev(Begin(), tobePrev) == true);
      }
     #ENDIF
     #IF (Cond2) //mutually exclusive with two-way
      {  tobeNext = Begin();
         Post(checkNext(End(), tobeNext) == true);
         }
     #ENDIF
   }

 case "Non-Circular":
   plug-in: VP4-2 //container does not have a circular structure
   Cond1: ("VP5 == Two-way")
   Cond2: ("VP5 == One-way")
```

```
   Refine-a: Responsibility
      Void Push_front(tItem item),  Void Push_back(tItem item), tIterator
   Insert(tIterator pos, tItem item), tIterator InsertAfter(tIterator pos,
   tItem item), Void Pop_front(), Void Pop_back(), Void Erase(tIterator pos)
   {
     #IF (Cond1)
      {  tobeNext = null;
         Post(checkNext(End(), tobeNext) == true);
         tobePrev = null;
         Post(checkPrev(Begin(), tobePrev) == true);
      }
     #ENDIF
     #IF (Cond2)//mutually exclusive with two-way
      {  tobeNext = null;
         Post(checkNext(End(), tobeNext) == true);
         }
     #ENDIF
   }
}

Variation VP5 <ReverseType>[1..1] outof 2
/* This variability contract illustrates the flexibility of actions in such
   contracts. Here, the responsibility find_prev is declared differently
   depending on whether the container is bi-directional or not.
   If it is, then we obsvervability find_prev that requires an Iterator.
   Otherwise, an Iterator going to the previous element is NOT possible
   and thus we must define a new responsibility Previous (to be bound to
   some procedure of the IUT, that somehow returns the previous element.
   It must be emphasized that the domain contract uses the abstraction,
   that is, refers to find_Prev. Only once VP5 is handled can it be
   established how this abstraction is actually working.
*/
{
 case "Two-way":
   plug-in: VP5-1 //container has a reverse structure
   Refine-a: Declaration
   tIterator find_prev
   {
      Observability tIterator findPrev(tIterator pos);
   }

   Refine-a: Observability
   tIterator find_prev(tIterator pos)
   {
      value = findPrev(pos);
   }
 case "One-way":
   plug-in: VP5-2 //container does not have a reverse structure
   Refine-a: Declaration
   tIterator find_prev
   {
      Observability tIterator Previous(tIterator pos);
   }
   Refine-a: Observability
   tIterator find_Prev(tIterator pos);
   {
      value = Previous(pos);
   }
}

Variation VP6 <End-dummy>[1..2] outof 2
/* Different postconditions depending on presence or absence of dummy end
   This dummy end node, is after the last node containing an element.
```

```
   The observability End returns this node.

{
 case "True":
   plug-in: VP6-1 //End-dummy node is created
   Refine-a: Responsibility
   new()   //postconditions for the creation of a dummy node
   {
     Post(End() not= null );
     Post(itemAt(End()) == null);
     Post(size() == 1);
   }
 case "False":
   plug-in: VP6-2 //End-dummy is not created
   Refine-a: Responsibility
   new()   //postconditions if there is no dummy node
   {
     Post(End() == null);
     Post(size() == 0);
   }
}

Variation VP7 <Random-Access> [1..1] outof 2
/* Typical generative approach la Czarnecki and Eisenecker:
   If VP7-1, then we define observability check_itemAt
   otherwise it is left empty.
   Most importantly, in the case of an indexable container of
   fixed size supporting random access, we use operatorIndex(n)
   which is relevant to VP8.
*/
{
  case "True":
   plug-in: VP7-1 //Container has random access via indexes
   Cond1: ("VP8 == Indexable and VP1 == Fixed")
   Refine-a: Observabibility
   Boolean check_itemAt(Integer n)
   {
      #IF (Cond1)
      value = (operatorIndex(n) == itemAt(Begin() + n));
      #ENDIF
   }
  case "False":
   plug-in: VP7-2 //no random access
   Refine-a: Observabibility
   Boolean check_itemAt(Integer n)
   {}

}


Variation VP8 <IndexingType> [1..1] outof 2
/* If dealing with an Arrayed list, then the observability
   operatorIndex must be added to the domain contract.
   The key point to understand is that VP7 uses operatotIndex
   for one specific configuration of variants.
   But it's VP8 that modifies the domain contract to declare
   this observability.
*/
{
  case "Indexable":
   plug-in: VP8-1 //Container is an Arrayed-list
   cond1: ("VP1 == Fixed")
   Refine-a: Declaration
```

```
        tItem operatorIndex
            {
            #IF (cond1)
               Observability tItem operatorIndex(Integer index);
               #ENDIF
        }
      case "Non-Indexable":
        plug-in: VP8-2 //Container is a Linked-list
        Refine-a: Declaration
        tItem operatorIndex
        {}
    }

    Variation VP30 <Length-counter> [1..1] out of 2
-   /* This variation point is an optional feature with one variant VP31 "LC-Type"
    [CZA99]. As it was explained in subsection 3.2, we have added another
    variant to this variation point named VP30-2 "NO-Lcounter". In domain
    contracts, the return type for observabailities size( ) and max_size( ) is a
    generic type "INT" that can vary among the integer types. In case of
    selecting variant VP30-2, "INT" can conform to the default "Integer" type.
    And in case of selecting any of three variants of VP31 (below), "INT" can
    conform to "int" or "short" or "long" types.
    This variability depends to the existence of VP1 (regardless of any variants
    for VP1) according to RT table. This constraint is added to the contract as
    cond1. The refinement for this variation point is specified in the Exports
    section of the domain contracts.
    */
    {
      case "LC-Type":
        Plug-in: VP31 //LC-Type is defined by VP31
        Cond1: ("VP1 == Variable or VP1 == Fixed")
            Refine-a: Exports
        Type length-counter
        {  #IF (Cond1)
            " "
           #ENDIF
        }
      case "No-Lcounter":
        Plug-in: VP30-2 //LC-Type is default Integer
        Cond1: ("VP1 == Variable or VP1 == Fixed")
        Refine-a: Exports
        Type length-counter
        { #IF (Cond1)
              Type INT conforms Integer;
          #ENDIF
        }
    }

    Variation VP31 <LC-Type> [1..1] out of 3
    /* This variability contract is straightforward:
       The domain contract refers to INT as the type for the length counter
       if there's one.
       This variability contract simply refines this INT type to the exact
       type that must be used, depending on the selected variant.
    */
    {
      case "int":
        Plug-in: VP31-1 //length type is int
        Cond1: ("VP30 == LC-Type and (VP1 == Variable or VP1 == Fixed)")
            Refine-a: Exports
        Type LC-Type
        {
           #IF (Cond1)
```

```
        Type INT conforms int;
        #ENDIF
    }
  case "short":
    Plug-in: VP31-2 //length type is short
    Cond1: ("VP30 == LC-Type and (VP1 == Variable or VP1 == Fixed)")
    Refine-a: Exports
    Type LC-Type
    {
        #IF (Cond1)
            Type INT conforms short;
        #ENDIF
    }
  case "long":
    Plug-in: VP31-3 //length type is long
    Cond1: ("VP30 == LC-Type and (VP1 == Variable or VP1 == Fixed)")
    Refine-a: Exports
    Type LC-Type
    {
        #IF (Cond1)
            Type INT conforms long;
        #ENDIF
    }
}


Variation VP33 <Tracing> [1..1] out of 2
{
  case "On":
    Plug-in: VP33-1 //create forward links from RT table
        Refine-a: Declaration
    Void Dump-Forward-links
        {
        Observability Void Dump-Forward-links();
    }
  case "Off":
    Plug-in: VP33-2 //no tracing links are needed
    Refine-a: Declaration
    Void Dump-Forward-links
        {}
}
```

(The grammar for variability contracts is given in Appendix A2.)

The variability contracts constitute the core of our generative approach. In particular, the one for VP3 illustrates the ability to refine an element of the domain contract or, conceptually, take it out of the latter. VP4 provides an example of the general mechanism we support to associate an action to a configuration of variants from different variation points. And VP5 demonstrate how an abstraction used in a domain contract (such as finding the previous element in a container) can be refined to different specific mechanisms in a particular member contract depending on the exact selection of variants. The point to be emphasized is that the proposed variability

contracts provide a general mechanism for the refinement of a domain artifact: only the definition of the actions is specific to ACL. Should the domain contract consist in some other type of textual specification (e.g., a Spec Explorer one), our generative mechanism, which rests on the creation of the relationship table and its 'transformation' into these variability contracts, would remain unaltered. This, we believe, is a key contribution of this work. Moreover, in contrast to approaches such as the "Consolidated Variability Model" (CVM) [BAY06], "Orthogonal Variability Model" (OVM) [POH05] or "A family of languages for Variability Management in Software Product Lines" (VML*) [ZSCH09], there is a direct correspondence between our relationship table and such variability contracts:

- the RT defines a feature grammar, which proceeds from the feature diagram but is not dependent on the specification of other domain artifacts (such as our domain contracts).

- the variability contracts integrate the information of the RT table (possibly combining several rows into a single contract as shown above) with actions specific to the domain artifacts.

As mentioned in chapter 1, the process of going from the relationship table to the variability contracts is currently performed manually. However, we must emphasize that this process is systematic inasmuch as it proceeds from the following algorithm (whose automation has been assigned to another student of our research group): the inputs are variation points, variants and variability relationships from feature model and RT table, and the outputs are variability contracts in ACL-V:

```
1.   for each Variation VPx with x in [0..n] from feature diagram
2.     Specify: "Variation" code "<"name">" "["option"]" "out-of"out-of
3.     for each variant in Variation VP-x
4.       Specify: "case" "variant-name:"
5.       Specify: "plugin:" VPx-m
6.       if variation VPx involves with conditions in RT
7.        Define conditions according to Implementation Rules for variability
          relationship such as cond1,cond2,etc
8.        for each defined condition
9.          Specify: "Cond#:" "(" cond# ")"
10.       endfor
```

```
11.       endif
12.       Search the domain contracts to find the Refinement-artifacts:
           Variabilities, Observabilities, Responsibilities, Scenarios,
           Exports, and etc
13.     Identify the potential point(s) that variability can occur
14.           Mark all the potential point(s) with a comment //plugin_VPx()
15.              Set Intra-Variant# with potential point(s)#
16.    for each Intra-Variant#
17.     Specify: "Refine-"Refinement-type ":" Refinement-artifact
18.     Specify: artifact-haeder as: artifact-type artifact-name "{"
19.      if Variation VPx involves with conditions in RT
20.        for each defined condition
21.         Specify: "#IF" "(" Cond# ")"
22.         Specify: "{"contract body in ACL with respect to Cond#"}"
23.         Specify: "#ENDIF"
24.        endfor
25.      else
26.              Specify: contract body in ACL
27.      endif //RT condition
28.   Specify: "}"
29.   endfor //each Intra-Variant#
30. endfor //each variant
31. endfor //each variation point
```

This algorithm presupposes that the domain engineer has identified which individual rows of the relationship table (RT) must be integrated into what we call an *intra-Variant* (will be explained shortly). For an example, see VP4 above.

More on Intra-Variants: In a generative process, selecting a specific variant can cause various changes on different artifacts in the domain model, and each of these changes may need a different format to be specified. These variability relationships and their affects, either directly or indirectly, should be considered and specified in the domain model in order to provide proper information for deriving a member from the domain. On this respect, we have defined the concept of "*intra-Variant*" for a variant, which introduces all the possible *variations* under that variant. Each *intra-Variant* indicates a possible affect of its variant on a specific artifact of the domain.

Finally, the variability contracts associate a label (called a plug-in) with each configuration that requires an action. Such labels are the links between the variability contracts and the domain contracts as will now be explained in the next phase.

5 Phase III: Selection Templates and Contract Repository

Phase III pertains to the creation of the XML artifacts used as input files by our generative process. The first of these files is the one for the variability contracts presented above. This is a straightforward transformation, which we now briefly explain:

5.1 The Variability Contract Repository

XML is a descriptive meta-language, which makes it possible to describe the user-defined tags. An xml document forms a tree structure. Each xml document contains one or more elements, bounded in a start- and an end- tag. The text between the start- and end- tags is called the element's content, and the name in the start- and end- tags identifies the element's type. Each element may have one or more attributes, and each attribute has a name and an assigned value.

The transformation from the above variability contracts to their xml equivalents proceeds from the following guidelines:

- Element type "Variation-x": defines the start "<" and end "/>" tags of this variation point with "Variation-x", where x is the number assigned to this variation point

- Attribute "name": defines the name of this variation point, which is the same as the name presented in the feature diagram

- Attribute "code": defines an identifier (which we call a code) for Variation-x as VPx, which is used in configuration list

- Attribute "contract-name": defines the name of the contract(s) in the domain contracts that is affected by this variability

- Attribute "option": defines that how many variants should be bound to this variation point. This number comes from the relationship between the variants and their variation point.

- Attribute "out-of": defines that how many variants exist for this variation point. For example if the variation point has 4 different variants the attribute "out-of" is "4" and if they are all of type exclusive-or then the attribute "option" is "1", which means that only one of these variants should be selected to be bound to this variation point.

Then in each variation-x element, there will be sub-elements corresponding to its variants with the following attributes. The number of these sub-elements is equal to the number of attribute "out-of" in the variation-x:

- Element type "<" "/>": defines the start and end tags of each variant with "Case"

- Attribute "name": defines the name of the variant, which is the same as the name presented in the feature diagram and variability contract

- Attribute "cond1": contains the condition for a relationship between this variation point and any other ones, if there is any in the relationships table (RT); if there is no relation between this variation point and any other ones, this attribute should be empty " " or eliminated

- Attributes "cond-x" and their relevant "variant-x" can be added to this "Case", if there are more variants related to this variation point in RT. It should be noted that all of the related variants must already be recorded in the RT with their relational conditions in the "Condition Part" and results in the "Action Part".

- Attribute "Rule": indicates the number of the variability relation rules that forms "cond1" and "variant1". If cond1 is a compound condition formed from different rules, it is the number of the first rule.

- Attribute "Rule-x" is the number of the other variability relational rules that form "cond-x" and their relevant "variant-x".

Whereas all of this is straightforward, the specification of intra-variants is slightly more complex. In essence, an intra-variant line must be created for each separate action

associated with a particular configuration of variants. The transformation from the variability contract is still direct. Each "intra-Variant" sub-element can specify some attributes for the required plug-ins. These attributes are as follows:

- Sub-element type "<" "/>": defines the start and end tags of each "intra-Variant"

- Attribute "code": defines that how the variability will affect the domain contracts.

- Attribute "type": defines where the variability will affect the domain contract.

- Attribute "name": defines which contract item will exactly be affected by the variability;.

- Attribute "variant": contains the body of the plug-in item if there is no condition, that is, if the variant is empty or " ".

- Attribute "variant-x": contains the "Action Part" in contract format if there is a condition in attribute "cond-x" above. This will be the body of the plug-in item. If there is no condition in attribute "cond-x", then this attribute should be empty " " or eliminated.

The algorithm used to obtain the specification of each intra-variant in the repository follows: the input is variability contract repository (after transformation from phase II to phase III), and the output is plugin labels in domain contracts:

```
1.  Start from the beginning of the variability contracts repository
2.  for each Variation-x with x in [0..n]
3.    for each intra-Variant or variant Case
4.    if plugin-code is "Add"
5.       if plugin-type is "Declaration" or "Exports"
6.          within domain contracts with contract-name
7.          go to the end of Declaration or Exports part
8.          create label plugin_VPx() and add the label
9.       endif
10.      if plugin-type is "Observability" or "Responsibility"
11.         within domain contracts with contract-name
12.         find the Observability with plugin-name
13.               go to the end of the context
14.         create label plugin_VPx() and add the label
15.      endif
```

```
16.  else
17.  if plugin-code is "Replace"
18.     if plugin-type is "Observability" or "Responsibility"
19.        within domain contracts with contract-name
20.        find the Observability or Responsibility with plugin-name
21.        find the "marks" for plugin in the context
22.        create label plugin_VPx() and replace it with the "marks"
23.     endif
24.     else
25.     if plugin-code is " "
26.       no label plugin is needed
27.     endif
28.  endfor //intra-Variant or variant Case
29. endfor //Variation-x
```

Again, we remark that another student of our research group is currently investigating the automation of this transformation, which rests on this algorithm.

All of the above discussion is clarified and illustrated by considering the end result for the variability contracts given above. Consequently, we include below the actual complete xml file used for generating member contracts for the domain of sequential list-based containers.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="birds.xsl"?>


<Contracts>
    <Variation0
     name="is-key-based"  code="VP0"   contract-name="MainContainer"  option="1" out-of="2">
        <Case    name="True">
            <intraVariant  code="Add"   type="Exports" name="key_type"
                    variant="          Type key_type conforms Key;"/>
        </Case>
        <Case    name="False">
            <intraVariant  code="Add"   type="Exports" name="key_type"
                    variant=" "/>
        </Case>
    </Variation0>

    <Variation1
     name="LengthType"   code="VP1"   contract-name="ContainerBase"  option="1" out-of="2">
        <Case    name="Variable">
            <intraVariant  code="Add"   type="Observability"name="IsFull"
                    variant="value = false;"/>
        </Case>
        <Case    name="Fixed">
            <intraVariant  code="Add"   type="Observability"name="IsFull"
                    variant="value = (size() == max_size());"/>
        </Case>
    </Variation1>

    <Variation2
     name="StorageType"  code="VP2"   contract-name="MainContainer"  option="1" out-of="2">
        <Case    name="Copy">
            <intraVariant  code="Add"   type="Responsibility"     name="Pop_front"
                    variant=" "/>
            <intraVariant  code="Add"   type="Responsibility"     name="Pop_back"
                    variant=" "/>
```

```
        <intraVariant  code="Add"   type="Responsibility"    name="Erase"
                variant=" "/>
    </Case>
    <Case   name="Reference">
        <intraVariant  code="Add"   type="Responsibility"    name="Pop_front"
                variant="Post(pos == null);"/>
        <intraVariant  code="Add"   type="Responsibility"    name="Pop_back"
                variant="Post(pos == null);"/>
        <intraVariant  code="Add"   type="Responsibility"    name="Erase"
                variant="Post(pos == null);"/>
    </Case>
</Variation2>

<Variation3
  name="Resizable"     code="VP3"   contract-name="MainContainer"  option="1" out-of="2">
    <Case   name="True"  cond1="VP1 = Fixed"    Rule1="5">
        <intraVariant  code="Add"   type="Responsibility"    name="resize"
                variant1="Pre(size() == max_size());
                    &#10;          old_Size = size();
                    &#10;          Execute();
                    &#10;          Post(max_size() == old_Size * 2);"/>
    </Case>
    <Case   name="False">
        <intraVariant  code="Add"   type="Responsibility"    name="resize"
                 variant=" "/>
    </Case>
</Variation3>

<Variation4
  name="RotateType"   code="VP4"   contract-name="MainContainer"  option="1" out-of="2">
    <Case   name="Circular"cond1="VP5 = Two-way"cond2="VP5 = One-way"Rule1="5" Rule2="5">
        <intraVariant  code="Add"   type="Responsibility"    name="Push_front"
                variant1="tobeNext = Begin();
                    &#10;          Post(checkNext(End(), tobeNext) == true);
                    &#10;          tobePrev = End();
                    &#10;          Post(checkPrev(Begin(), tobePrev) == true);"
                variant2="tobeNext = Begin();
                    &#10;          Post(checkNext(End(), tobeNext) == true);"/>
        <intraVariant  code="Add"   type="Responsibility"    name="Push_back"
                variant1="tobeNext = Begin();
                    &#10;          Post(checkNext(End(), tobeNext) == true);
                    &#10;          tobePrev = End();
                    &#10;          Post(checkPrev(Begin(), tobePrev) == true);"
                 variant2="tobeNext = Begin();
                    &#10;          Post(checkNext(End(), tobeNext) == true);"/>
        <intraVariant  code="Add"   type="Responsibility"    name="Insert"
                variant1="tobeNext = Begin();
                    &#10;          Post(checkNext(End(), tobeNext) == true);
                    &#10;          tobePrev = End();
                    &#10;          Post(checkPrev(Begin(), tobePrev) == true);"
                 variant2="tobeNext = Begin();
                    &#10;          Post(checkNext(End(), tobeNext) == true);"/>
        <intraVariant  code="Add"   type="Responsibility"    name="InsertAfter"
                variant1="tobeNext = Begin();
                    &#10;          Post(checkNext(End(), tobeNext) == true);
                    &#10;          tobePrev = End();
                    &#10;          Post(checkPrev(Begin(), tobePrev) == true);"
                 variant2="tobeNext = Begin();
                    &#10;          Post(checkNext(End(), tobeNext) == true);"/>
        <intraVariant  code="Add"   type="Responsibility"    name="Pop_front"
                variant1="tobeNext = Begin();
                    &#10;          Post(checkNext(End(), tobeNext) == true);
                    &#10;          tobePrev = End();
                    &#10;          Post(checkPrev(Begin(), tobePrev) == true);"
                 variant2="tobeNext = Begin();
                    &#10;          Post(checkNext(End(), tobeNext) == true);"/>
        <intraVariant  code="Add"   type="Responsibility"    name="Pop_back"
                variant1="tobeNext = Begin();
                    &#10;          Post(checkNext(End(), tobeNext) == true);
                    &#10;          tobePrev = End();
```

```
                    &#10;          Post(checkPrev(Begin(), tobePrev) == true);"
                variant2="tobeNext = Begin();
                    &#10;          Post(checkNext(End(), tobeNext) == true);"/>
        <intraVariant  code="Add"   type="Responsibility"    name="Erase"
                variant1="tobeNext = Begin();
                    &#10;          Post(checkNext(End(), tobeNext) == true);
                    &#10;          tobePrev = End();
                    &#10;          Post(checkPrev(Begin(), tobePrev) == true);"
                variant2="tobeNext = Begin();
                    &#10;          Post(checkNext(End(), tobeNext) == true);"/>
    </Case>
    <Case name="Non-Circular" cond1="VP5 = Two-way"cond2="VP5 = One-way" Rule1="5"Rule2="5">
        <intraVariant  code="Add"   type="Responsibility"    name="Push_front"
                variant1="tobeNext = null;
                    &#10;          Post(checkNext(End(), tobeNext) == true);
                    &#10;          tobePrev = null;
                    &#10;          Post(checkPrev(Begin(), tobePrev) == true);"
                variant2="tobeNext = null;
                    &#10;          Post(checkNext(End(), tobeNext) == true);"/>
        <intraVariant  code="Add"   type="Responsibility"    name="Push_back"
                variant1="tobeNext = null;
                    &#10;          Post(checkNext(End(), tobeNext) == true);
                    &#10;          tobePrev = null;
                    &#10;          Post(checkPrev(Begin(), tobePrev) == true);"
                variant2="tobeNext = null;
                    &#10;          Post(checkNext(End(), tobeNext) == true);"/>
        <intraVariant  code="Add"   type="Responsibility"    name="Insert"
                variant1="tobeNext = null;
                    &#10;          Post(checkNext(End(), tobeNext) == true);
                    &#10;          tobePrev = null;
                    &#10;          Post(checkPrev(Begin(), tobePrev) == true);"
                variant2="tobeNext = null;
                    &#10;          Post(checkNext(End(), tobeNext) == true);"/>
        <intraVariant  code="Add"   type="Responsibility"    name="InsertAfter"
                variant1="tobeNext = null;
                    &#10;          Post(checkNext(End(), tobeNext) == true);
                    &#10;          tobePrev = null;
                    &#10;          Post(checkPrev(Begin(), tobePrev) == true);"
                variant2="tobeNext = null;
                    &#10;          Post(checkNext(End(), tobeNext) == true);"/>
        <intraVariant  code="Add"   type="Responsibility"    name="Pop_front"
                variant1="tobeNext = null;
                    &#10;          Post(checkNext(End(), tobeNext) == true);
                    &#10;          tobePrev = null;
                    &#10;          Post(checkPrev(Begin(), tobePrev) == true);"
                variant2="tobeNext = null;
                    &#10;          Post(checkNext(End(), tobeNext) == true);"/>
        <intraVariant  code="Add"   type="Responsibility"    name="Pop_back"
                variant1="tobeNext = null;
                    &#10;          Post(checkNext(End(), tobeNext) == true);
                    &#10;          tobePrev = null;
                    &#10;          Post(checkPrev(Begin(), tobePrev) == true);"
                variant2="tobeNext = null;
                    &#10;          Post(checkNext(End(), tobeNext) == true);"/>
        <intraVariant  code="Add"   type="Responsibility"    name="Erase"
                variant1="tobeNext = null;
                    &#10;          Post(checkNext(End(), tobeNext) == true);
                    &#10;          tobePrev = null;
                    &#10;          Post(checkPrev(Begin(), tobePrev) == true);"
                variant2="tobeNext = null;
                    &#10;          Post(checkNext(End(), tobeNext) == true);"/>
    </Case>
</Variation4>

<Variation5
  name="ReverseType"  code="VP5"   contract-name="MainContainer"   option="1"     out-of="2">
    <Case    name="Two-way">
        <intraVariant        code="Add"   type="Declaration"       name="find_prev"
                variant="Observability tIterator findPrev(tIterator pos);"/>
        <intraVariant        code="Add"   type="Observability"name="find_prev"
```

```
                variant="value = findPrev(pos); "/>
    </Case>
    <Case    name="One-way">
        <intraVariant        code="Add"    type="Declaration"        name="find_prev"
                variant="Observability tIterator Previous(tIterator pos);"/>
        <intraVariant        code="Add"    type="Observability"name="find_prev"
                variant="value = Previous(pos); "/>
    </Case>
</Variation5>

<Variation6
  name="end-dummy"    code="VP6" contract-name="MainContainer"   option="1"     out-of="2">
    <Case    name="True">
        <intraVariant   code="Add"    type="Responsibility"     name="new"
                variant="Post(End() not= null);
                    &#10;            Post(itemAt(End()) == null);
                    &#10;            Post(size() == 1);  "/>
    </Case>
    <Case    name="False">
        <intraVariant   code="Add"    type="Responsibility"     name="new"
                variant="Post(End() == null);
                    &#10;            Post(size() == 0);"/>
    </Case>
</Variation6>

<Variation7
  name="Random-Access"    code="VP7" contract-name="MainContainer"       option="1"out-of="2">
    <Case    name="True"   cond1="VP8 = Indexable  and  VP1= Fixed"   Rule1="5">
        <intraVariant   code="Add"    type="Observability"name="check_itemAt"
                variant1="value = (operatorIndex(n) == itemAt(Begin() + n)); "/>
    </Case>
    <Case    name="False">
        <intraVariant   code="Add"    type="Observability"name="check_itemAt"
                variant=" "/>
    </Case>
</Variation7>

<Variation8
  name="IndexingType"  code="VP8"    contract-name="MainContainer"  option="1" out-of="2">
    <Case    name="Indexable"        cond1="VP1 = Fixed"     Rule1="5">
        <intraVariant   code="Add"    type="Declaration"  name="operatorIndex"
                variant1="Observability tItem operatorIndex(Integer index);"/>
    </Case>
    <Case    name="Non-Indexable">
        <intraVariant   code="Add"type="Declaration"      name="operatorIndex"
                variant="  "/>
    </Case>
</Variation8>

<Variation30
  name="Length-counter"      code="VP30"  contract-name="MainContainer"  option="1"out-of="2">
    <Case    name="LC-Type"    cond1 = "VP1 = Variable  or   VP1 = Fixed" Rule1="9">
        <intraVariant   code="Add"    type="Exports" name="length-counter"
                variant1=" "/>
    </Case>
    <Case    name="No-Lcounter" cond1 = "VP1 = Variable  or   VP1 = Fixed"Rule1="9">
        <intraVariant   code="Add"    type="Exports" name="length-counter"
                variant1="            Type INT conforms Integer; "/>
    </Case>
</Variation30>

<Variation31
  name="LC-Type"  code="VP31"   contract-name="MainContainer"    option="1" out-of="3">
<Case    name="int"cond1="VP30 = LC-Type  and (VP1 = Variable  or   VP1 = Fixed)"    Rule1="7">
         <intraVariant  code="Add"    type="Exports" name="LC-Type"
                variant1="           Type INT conforms int; "/>
    </Case>
<Case    name="short"cond1="VP30 = LC-Type and (VP1 = Variable  or   VP1 = Fixed)"  Rule1="7">
        <intraVariant   code="Add"    type="Exports" name="LC-Type"
                variant1="           Type INT conforms short; "/>
```

```
        </Case>
    <Case    name="long"cond1="VP30 = LC-Type and VP1 = Variable  or  VP1 = Fixed)"   Rule1="7">
            <intraVariant  code="Add"   type="Exports" name="LC-Type"
                    variant1="          Type INT conforms long; "/>
        </Case>
    </Variation31>

    <Variation33
      name="Tracing"  code="VP33"  contract-name="MainContainer"    option="1" out-of="2">
    <Case    name="On" >
            <intraVariant  code="Add"   type="Declaration"  name="Dump-Forward-links"
                    variant="Observability Void Dump-Forward-links(); "/>
        </Case>
        <Case    name="Off" >
            <intraVariant  code="Add"   type="Declaration"  name="Dump-Forward-links"
                    variant=" "/>
        </Case>
    </Variation33>
</Contracts>
```

5.2 From Domain Contracts to Variability Selection Templates

The variability contracts establish what actions are to take place in the domain contracts when a particular configuration of variants is present in the configuration list. As previously mentioned, each such action is associated with a plug-in label. And as illustrated in the previous subsection, in order to transform the domain contracts into an xml file usable by our generative process, these labels must be correctly inserted in the domain contract, as illustrated in the abstract and main contract presented earlier in this chapter. The next step consists in producing an xml equivalent of the domain contracts, where, in order to enable generation, these plug-in labels are replaced by *variability selection templates* (since we are using a template based approach to generation). These templates are implemented in our solution using XSL style-sheets (XSLT) [XSL-a, XSL-b].

An XSL style-sheet (XSLT) is a rule-based file that consists of a set of rules for transforming a document expressed in xml. The rules in an xslt file are called *templates*. A template contains of *rules* and *patterns*. The template rules can be applied on an xml file when a specific node in that xml file is matched with the patterns. Such templates are of the general form:

```
<xsl:template match="pattern">
    body;
</xsl:template>
```

This command will search through the xml document and find all the elements whose name matches *pattern*. The body of the template contains text and/or instructions to extract and modify parts of the input that matches with the pattern. Without going into details, if more than one template rule matches, then the one with highest priority will be selected. An xslt *processor* then derives information from the xml file at hand according to the templates it instantiates, and then adds the results of this derivation into the generated output. The processor interprets the xslt file in a top-down manner.

The process of inserting such a template at a particular plug-in in the domain contract can be illustrated by considering the first plug-in label in the abstract contract presented earlier (see section 4.1) and repeated below:

```
Observability Boolean IsFull()
        {   //plugin VP1();
        }
```

This plug-in is augmented with a selection template as follows:

```
Observability Boolean IsFull()
    {
        </xsl:text>
        <xsl:apply-templates select="Variation1">
            <xsl:with-param name="param1" select="'Observability'" />
            <xsl:with-param name="param2" select="'IsFull'" />
        </xsl:apply-templates>
        <xsl:text disable-output-escaping="yes">
    }
```

In essence, </xsl:text> stops the generation process from treating the input as not requiring generation. In other words, we indicate we are stopping to process a common part of the domain contracts and entering a point where the text to output has to be generated proper. Then the select command requires that Variation1 be found and processed at this point of the file. Then the <xsl:text disable...> command indicates we go back to processing the text as being the common part of the domain contract.

All other plug-ins are dealt with in a similar way using the following algorithm (which is, again, under investigation for automation by another student): the input is domain contracts with plugin labels and the output is the same contracts with selection templates (part1) instead of plugin labels:

```
1.  Start from the beginning of the domain contracts file
2.  create a start tag to contain the common part of the contracts
3.  for each label plugin_VPx() with x in [0..n]
4.    remove the label or comment it out
5.    create an end tag to end the common part up to this point
6.    create a selection template to find "Variation-x" in the repository
7.       if the selection deals with an intra-Variant
8.          create the selection parameters with plugin-name and plugin-
    type
9.       endif
10.      create an end tag to end the selection template part
11.   create a start tag to resume the common part from this point
12.  endfor
13. create an end tag to end the common part of the contracts at end of the
    file
```

In details, the selection template created at line 6 replaces the plug-in label in domain contracts. It will then apply a template on the variability repository to find the particular variation point "Variation-x". Lines 2, 5 and 11 are related to common parts in the domain contracts. The common parts of the domain contracts will not be changed by the domain variability and they will be shared in all the generated outputs. For this reason, we just move them to the generated output as they are, by surrounding them by a start and an end tag. So before the selection template and after it, we create an end tag and resume the common parts (lines 5 and 11, respectively).

In order for these 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. This is best explained with an example. Consider, for example, VP4 in the variability contracts of section 4.2, which is the most complex of them. It involves applying some action (which depends on a particular combination of variants) to several responsibilities. In the domain contract, *each* of these responsibilities is augmented with a placeholder then replaced by an xslt selection template. In addition, at the end of the xml file for the domain contracts, the following xslt code must be added for VP4:

```
<xsl:template match="Variation4">
  <xsl:param name="param1"/>
  <xsl:param name="param2"/>
  <xsl:for-each select="Case">
    <xsl:choose>
    <xsl:when test="@name=$VP4  and  $VP4='Circular' and $VP5='Two-way'">
      <xsl:for-each select="intraVariant">
       <xsl:choose>
          <xsl:when test="@type=$param1 and @name=$param2">
              <xsl:value-of select="@variant1"/>
          </xsl:when>
          <xsl:otherwise>
          <xsl:text></xsl:text>
          </xsl:otherwise>
       </xsl:choose>
      </xsl:for-each>
    </xsl:when>
    <xsl:when test="@name=$VP4  and  $VP4='Circular' and  $VP5='One-way'">
      <xsl:for-each select="intraVariant">
       <xsl:choose>
          <xsl:when test="@type=$param1 and @name=$param2">
              <xsl:value-of select="@variant2"/>
          </xsl:when>
          <xsl:otherwise>
          <xsl:text></xsl:text>
          </xsl:otherwise>
       </xsl:choose>
      </xsl:for-each>
    </xsl:when>
    <xsl:when test="@name=$VP4  and  $VP4='Non-Circular' and  $VP5='Two-way'">
      <xsl:for-each select="intraVariant">
       <xsl:choose>
          <xsl:when test="@type=$param1 and @name=$param2">
              <xsl:value-of select="@variant1"/>
          </xsl:when>
          <xsl:otherwise>
          <xsl:text></xsl:text>
          </xsl:otherwise>
       </xsl:choose>
      </xsl:for-each>
    </xsl:when>
    <xsl:when test="@name=$VP4  and  $VP4='Non-Circular'  and  $VP5='One-way'">
      <xsl:for-each select="intraVariant">
       <xsl:choose>
          <xsl:when test="@type=$param1 and @name=$param2">
              <xsl:value-of select="@variant2"/>
          </xsl:when>
          <xsl:otherwise>
          <xsl:text></xsl:text>
          </xsl:otherwise>
       </xsl:choose>
      </xsl:for-each>
    </xsl:when>
    <xsl:otherwise>
          <xsl:text></xsl:text>
    </xsl:otherwise>
    </xsl:choose>
  </xsl:for-each>
</xsl:template>
```

In essence, this template command will start at the top level of the hierarchy in the variability repository in order to find an attribute that matches "Variation4". Then through the use of the *test* keyword, each possible configuration of variants handled by the variability contract is specified in order to determine which specific action of this

variability contract is to be used for the generation at that point of the domain contract. Each variation point requires a similar (albeit typically simpler) template command to 'hook' the actions of the variability contracts to the variation points marked in the domain contracts. The complete domain contracts augmented with such variability selection templates are given in Appendix A3.

Finally, let us conclude this description of Phase III by adding that the creation of these template commands proceeds from the following algorithm (which is under investigation by another student for possible automation): the input is domain contracts with selection templates (part1) and the output is the same contracts with selection templates (part1, part2):

```
1.Start at the end of the domain contracts
2.create a start tag to contain the selection templates for each Variation-x
3.for each Variation-x with x in [0..n] in the variability repository
4.   create instructions to define the input parameters for selection
5.   create a start tag to contain the selection templates for each variant
6.   for each variant in Variation-x
7.     create instructions to check the Variation-x and this variant in config
8.     if the Variation-x or this variant are involved with conditions in RT
9.       create the conditions and "and" them with instructions at step 14
10.      if conditions are formed by Rules5 or 6 in variability repository
11.        "and" the conditions with variant name
12.      endif
13.    endif
14.    create instructions to find the matched value for variant equal to the
       value defined in the configuration list "and" them with conditions
       defined at steps 8-12
15.    create instructions to select each intra-Variant
16.       for each intra-Variant in variant
17.         create instructions to check the input parameters   for selection if
              they are equal to the ones defined at part-1, in order
              to find the exact place-holder to plug-in
18.       create instructions to select the contracts of this variant
              related to conditions defined at steps 8-12 and 14
19.       create instructions to end the selection of each intra-Variants
20.        endfor //each intra-Variant
21.     endfor //each variant
22.     create an end tag to end the selection templates for each variant
23.  endfor //each Variation-x
24.  create an end tag to end the selection templates for each Variation-x
```

6 Phase IV: Generating and Validating a Member's Contract

6.1 Generation

At this point of the process, we want to apply a configuration list to our contracts in order to generate the contract for a specific member of the domain at hand (hereafter referred to a MC for member contract). The generative process starts with the compilation of the variability repository by a xml compiler (xerces [XML-a], [XML-b]) and of the domain contracts by an xslt compiler (xalan [XSL-a], [XSL-b]), as illustrated in Figure 6.1.

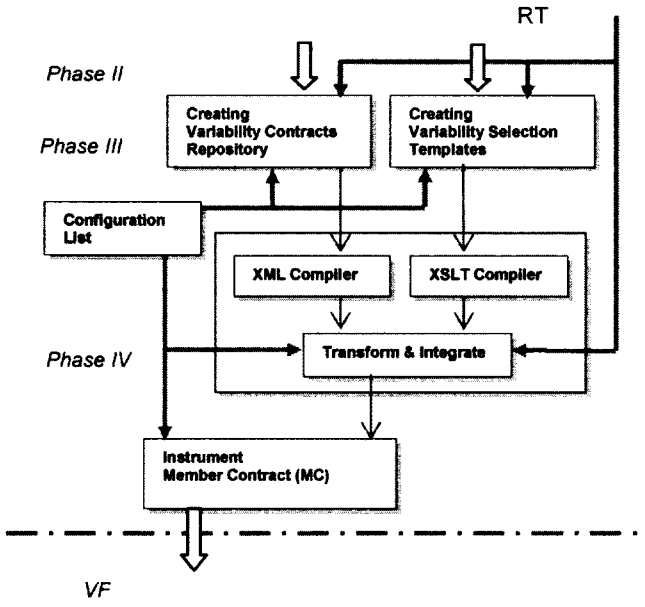

Figure 6.1 - Transformation from Phase III to Phase IV

Then, and only then, variability is resolved: by applying the configuration list on the variability repository and selecting the specific variants of the target MC, these variants can be integrated – that is, plugged – into the domain contracts at the appropriate location(s) of the domain contracts. This is achieved by a transformation (called "Transform & Integrate" in the figure above). This transformation (from phase III to phase IV), which we have fully implemented, realizes the following algorithm: the inputs are variability contracts repository, domain contracts with selection templates (part1, part2), configuration list to the transformer program, and the output is the desired member contract (Figure 6.1):

```
1. Start at the top level of the contract hierarchy in the domain contracts
2. Move the common part of the contracts to the output code
3. Apply the selection templates of "Variation-x" defined at part-1 of phase
   III, on the variability repository
4. for the matched element with "Variation-x" from the repository
5.         Apply the commands defined at part-2 of phase III
6.         Conduct the results to the output code
7. endfor
8. Repeat steps 2-7, until the end of the contract hierarchy in the domain
   contracts
```

where lines 3 to 5 are further refined as follows:

```
1. for each "Variation-x", with x in [0..n] that is matched with the
   selection template in the variability repository
2.       for each variant of "Variation-x"
3.     if variant is the same as the value defined for "Variation-x"
            in the configuration list of MC
4.             if "Variation-x" is involved in a variability relationship
               defined in RT, based on condition-A
5.                 if condition-A is true
6.                    go to check intra-variant (11)
7.                 else
8.                    go to the next Variation-x (20)
9.                 endif //condition-A
10.   endif //variant condition-A
11.        if variant deals with an intra-Variant
12.         for each intraVariant if parameters plugin-name and plugin type
                confirm the selection parameters
13.  select the variant definition from the repository, expand it in
                contract format, and integrate it with the domain
                contract at the point of the selection template
14.  endfor //each intra-Variant
15.        else
16.  select the variant definition for condition-A, expand it in contract
                format, and integrate it with the domain contract at the
                point of the selection template
17.  endif; //intra-Variant
18.      endif; //variant in config
19.   endfor //each variant
20. endfor //each variation-x
```

After this transformation, the generated MC does not contain any variability and is ready to act as a genuine input to ACL/VF. We discuss this validation step next.

6.2 Validation

6.2.1 About Traceability

Once the contract for a specific member of the domain at hand has been generated, it can be verified by inputting it to ACL/VF, compiling it, binding it to an implementation and then running this IUT in order to obtain a Contract Evaluation Report (as explained in chapter 1). Before doing so however, we suggest that the traceability that ought to exist between the different artifacts used by the generative process be verified. This verification process is being automated by the other member of our research group who is responsible for the automation of all transformation algorithms for Phases I, II and III. It is illustrated in Figure 6.2.

In This Figure the steps for creating and developing Traceability Links for Variability (TLV) in our generative framework is shown, as an extension to Figure 1.3 (our approach).

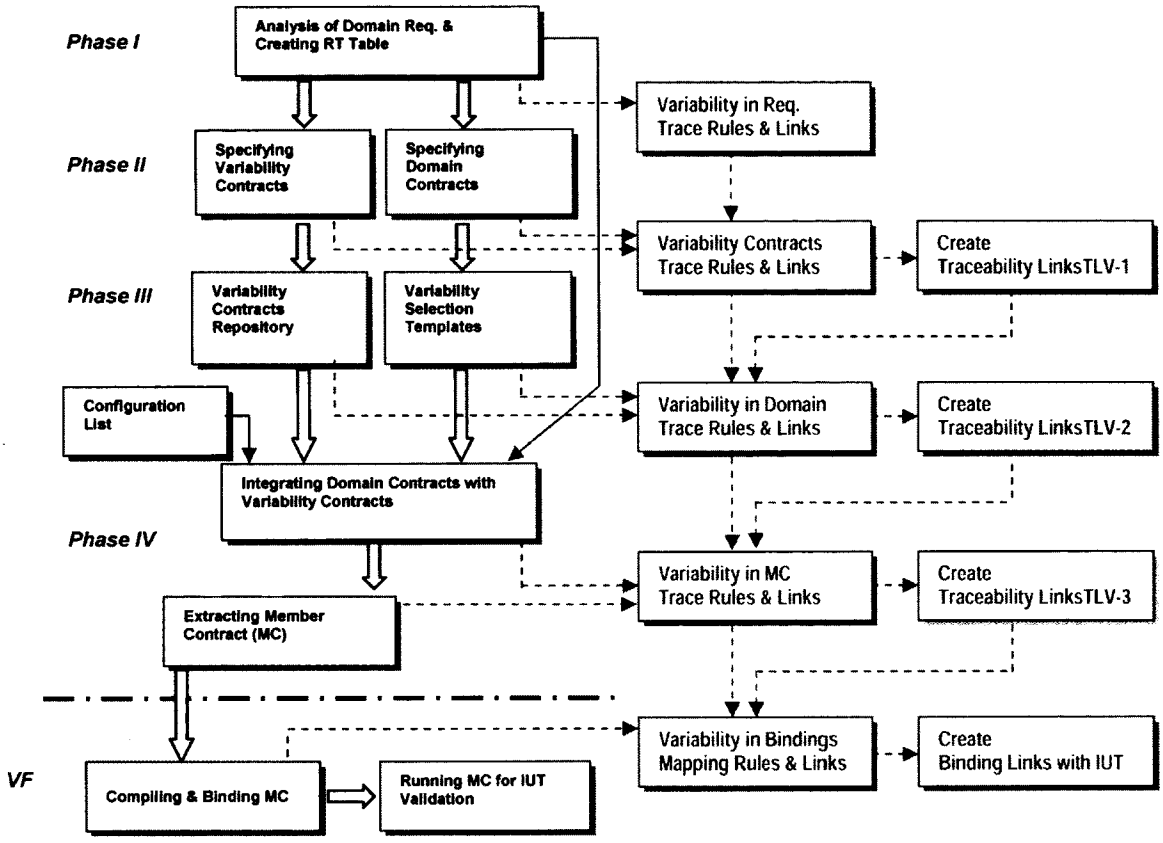

Figure 6.2 – An overview on Generative Framework and Traceability Links for Variability (TLV)

The nature of the traceability links developed in the additional tasks shown in Figure 6.2 is detailed in Figure 6.3.

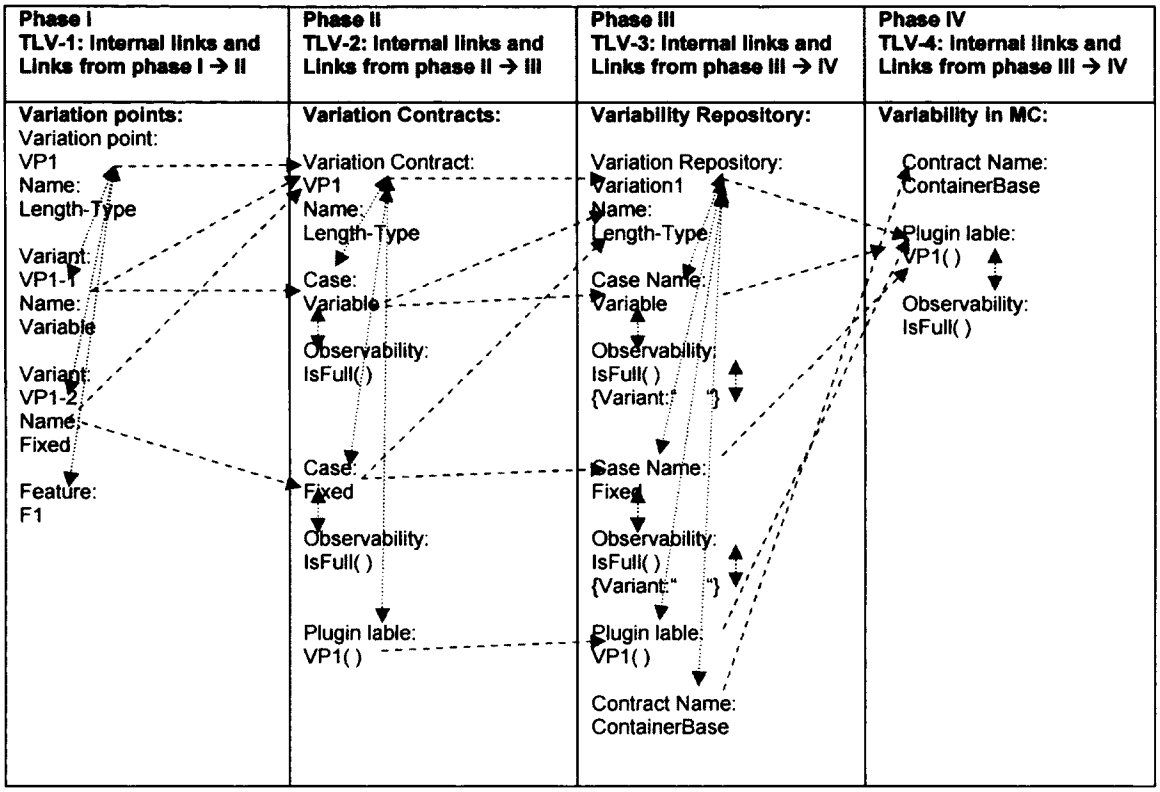

Figure 6.3 – Traceability links (forward) for related items at each phase and through different phases of our approach, only for Variation point 1 in the domain of Sequential Containers (List members)

In order to briefly overview this verification process, consider VP4 and VP5 of the first case study. In a nutshell: the fact that the relationships of the table of Phase I and the variability contracts of Phase II both have grammars (see first two appendices), allows us to verify not only their syntactic correctness but also their semantic correspondence. In figure 6.4, VP4 and VP5 are shown to trace to one entry of the relationship table. (They in fact trace to several rows of this table.) These two variation points are also shown to trace to specific variability contracts. The traceability verification process we envision (and carried out manually in our work) relies on such traceability links to ensure that all artifacts of our generative process semantically correctly 'map' onto each other so that all semantics pertaining to a specific variation point are consistent across these artifacts.

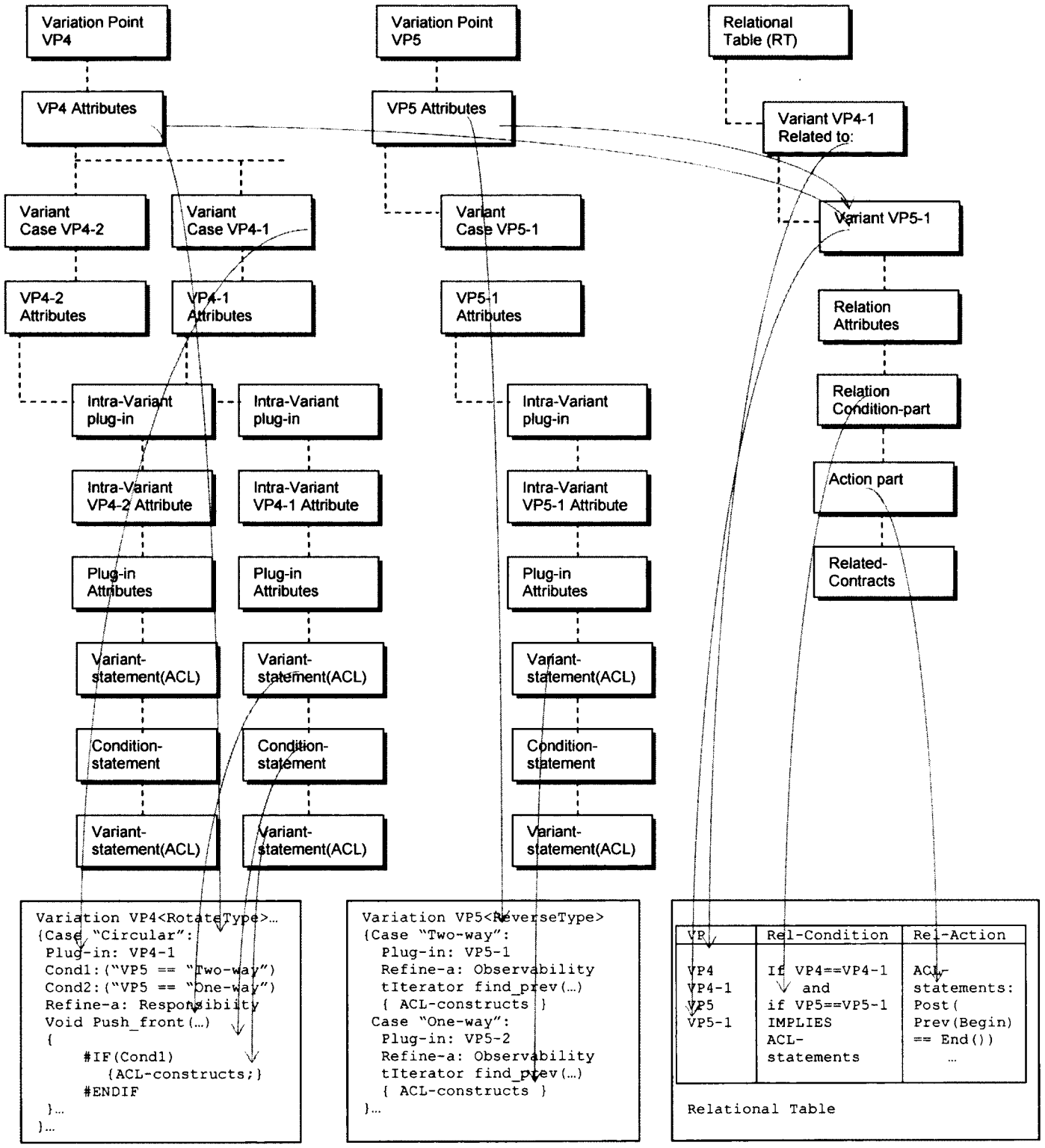

Figure 6.4 - Realization of Variability Contract Meta-model into Variability Contract Model in ACL-V and Relationships Table Meta-model into Relationships Table row for VP4 and VP5

6.2.2 Verifying a Member Contract

As previously mentioned, once the contract for a specific member of the domain at hand has been generated, it can be verified by inputting it to ACL/VF, compiling it, binding it to an implementation and then running this IUT in order to obtain a Contract Evaluation Report. Consider, for example, a single linked-list container. The configuration for it we use is:

```
<VP0 = False, VP1 = Variable, VP2 = Copy, VP3 = False,
 VP4 = Non-Circular, VP5 = One-way, VP6 = True, VP7 = False,
 VP8 = Non-Indexable, VP30 = No-LC-Type, VP31 = long, VP33 = off>
```

This configuration applied to our domain and variability contracts generates the contract given in Appendix 4. This contract can then be compiled in ACL, as shown in Figure 6.5.

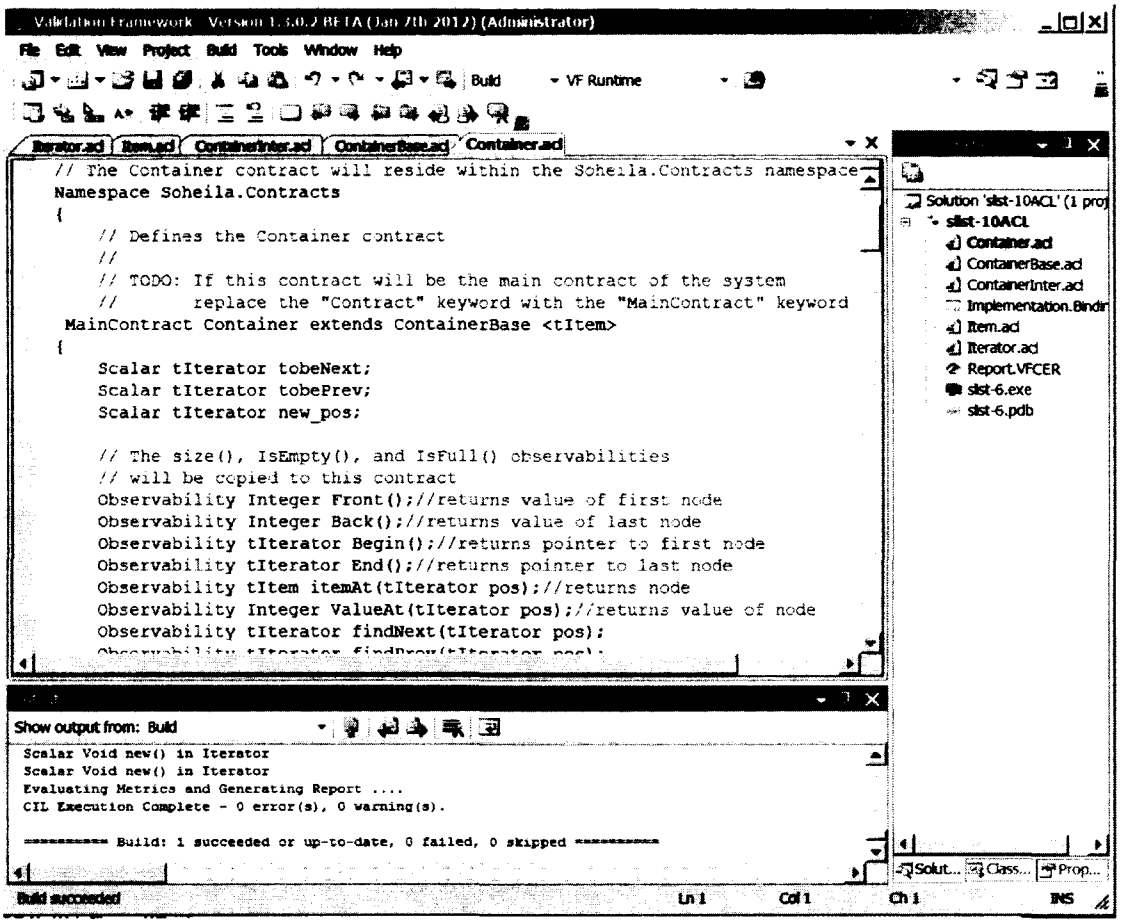

Figure 6.5 - Contracts Compilation Report for Member Contract: Single Linked-list

The generated contract is then bound to an actual implementation. This step (which is largely automated in ACL/VF via type inference) results in the binding table given below:

Definition of terms used in binding below (Table 6.1):

ACL Items type definition:
C = Contract R = Responsibility O = Observability E, T = Exported type

IUT Items type definition:
C = Class M = Method S = Structure T=type

Table 6.1 – Binding Table between Contract Items and their correspondent Items in C++ managed IUT, for the Domain of Sequential Containers (Single Linked-List Member)

ACL Items	ACL Statement (STL Sequential Container)	IUT Items	IUT Statement (STL Linked-list)
C	Item	C	Item
R	Item.new( )	M	Item::Item(Item^ nnext, int v)
R	Item.finalize( )	M	Item::~Item( )
O	Integer Item.Value( )	M	int Item::Value( )
C	Iterator	C	iterator
R	Iterator.new( )	M	Iterator::Iterator(Item^ x): node(x)
O	tItem getItem( );	M	Item^ Iterator::getItem( )
C	Container	C	Container
R	Container.new( )	M	Container::Container( ): length(0L)
R	Container.finalize( )	M	Container::~Container( )
O	tIterator Container.Begin( )	M	Iterator^ Container::Begin( )
O	Iterator Container.End( )	M	Iterator^ Container::End( )
O	Integer Container.Front( )	M	int Continer::Front( )
O	Integer Container.Back( )	M	int Container::Back( )
O	Boolean Container.isFull( )	M	bool Container::IsFull( )
O	Boolean Container.isEmpty( )	M	bool Container::IsEmpty( )
O	tItem Container.itemAt(tIterator pos)	M	Item^ Container::itemAt(Iterator ^pos)
O	Integer Container.Size( )	M	int Container::.Size( )
R	Void Container.Push_front(tItem item)	M	void Container::Push_front(Item ^item)
R	Void Container.Push_back(tItem item)	M	void Container::Push_back(Item ^item)
R	tIterator Container.Insert(tIterator pos, tItem item)	M	Iterator^ Container::Insert(Iterator^ pos, Item^ item)
R	tIterator Container.InsertAfter(tIterator pos, tItem item)	M	Iterator^ Container::InsertAfter(Iterator ^pos, Item ^item)
R	Void Container.Pop_front( )	M	void Container::Pop_front( )
R	Void Container.Pop_back( )	M	void Container::Pop_back( )
R	Void Container.Erase(tIterator pos)	M	void Container:: Erase(Iterator ^pos)
E	tItem	T	Container::Item
E	tIterator	T	Container::Iterator

Put simply, binding links the semantic elements of a contract (i.e., types, observabilities, responsibilities, etc.) of ACL to their corresponding counterparts in the implementation under test. Figure 6.6 shows an actual screen capture of this successful binding process for a single linked list.

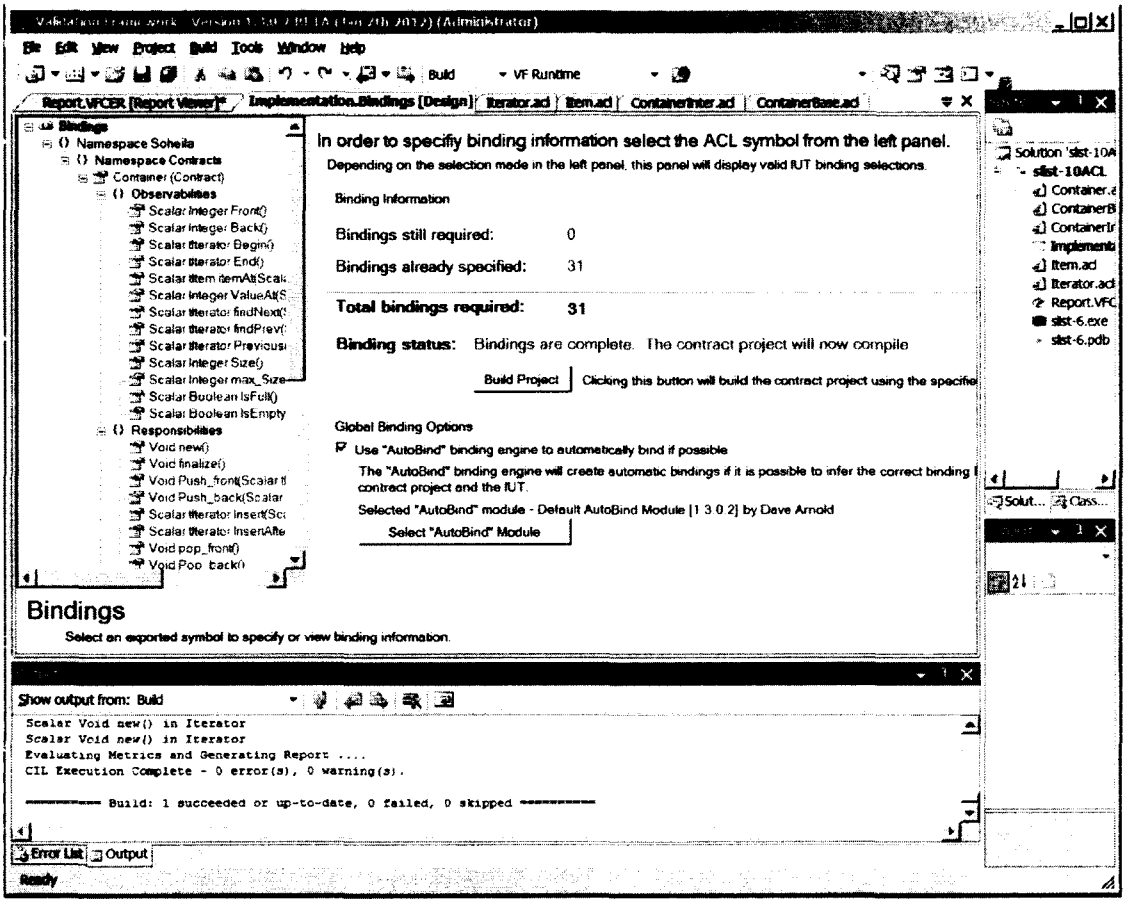

Figure 6.6 - Contracts Binding Report for Member Contract: Single Linked-list

Once this has been achieved, the generated contract is ready to be run against a particular test suite. A contract evaluation report is created for each execution of each implementation under test. Consequently, a faulty implementation can be quickly detected as illustrated in Figure 6.7 (in which several post-conditions and contracts have failed). (A particular responsibility can be scrutinized as shown in Figure 1.1)

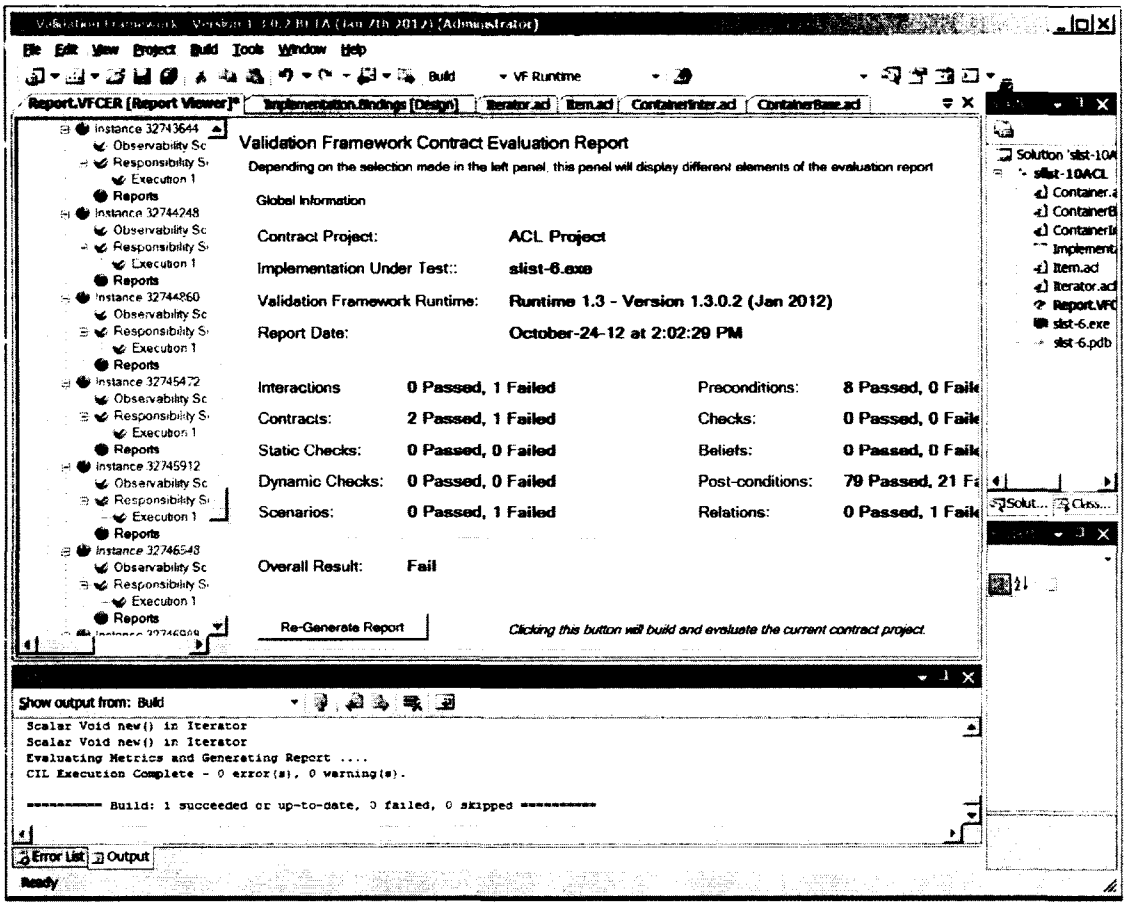

Figure 6.7 - Contracts Evaluation Report for Member Contract: Single Linked-list

Beyond testing several implementations against a generated contract, the domain itself can be extensively explored using ACL/VF by simply generating different members via different configurations. To conclude, here are some of the other configurations we have tested:

For a Double-linked-list:

<VP0 = False, VP1 = Variable, VP2 = Copy, VP3 = False,
VP4 = Non-Circular, VP5 = Two-way, VP6 = True, VP7 = False,
VP8 = Non-Indexable, VP30 = No-LC-Type, VP31 = long, VP33 = off>

For a Circular-single-linked-list:

<VP0 = False, VP1 = Variable, VP2 = Copy, VP3 = False, VP4 = Circular,
VP5 = One-way, VP6 = True, VP7 = False, VP8 = Non-Indexable,
VP30 = No-LC-Type, VP31 = long, VP33 = off>

For a Circular-double-linked-list:

<VP0 = False, VP1 = Variable, VP2 = Copy, VP3 = False, VP4 = Circular,
VP5 = Two-way, VP6 = True, VP7 = False, VP8 = Non-Indexable,
VP30 = No-LC-Type, VP31 = long, VP33 = off>

For completeness we include in Appendix 5 the generated contract for this last configuration (which the reader is invited to compare to the one for a single-linked list given in Appendix 4.)

7 A Second Case Study

We have developed a second case study to make sure that the generative process we had devised was not 'geared towards' a particular set of features. Furthermore, we wanted to explore how some features and contracts identified for sequential containers could be reused for a similar domain. Finally, we needed a domain for which an implementation would be ready to be bound to the contracts we would generate. The study of STL [STL95] suggested associated containers (i.e., containers that use a key) would be ideal to achieve our goals. Figure 7.1 shows how variation points of the sequential and associative domains were organized. Then the feature diagram specific to associative containers is given next in Figure 7.2.

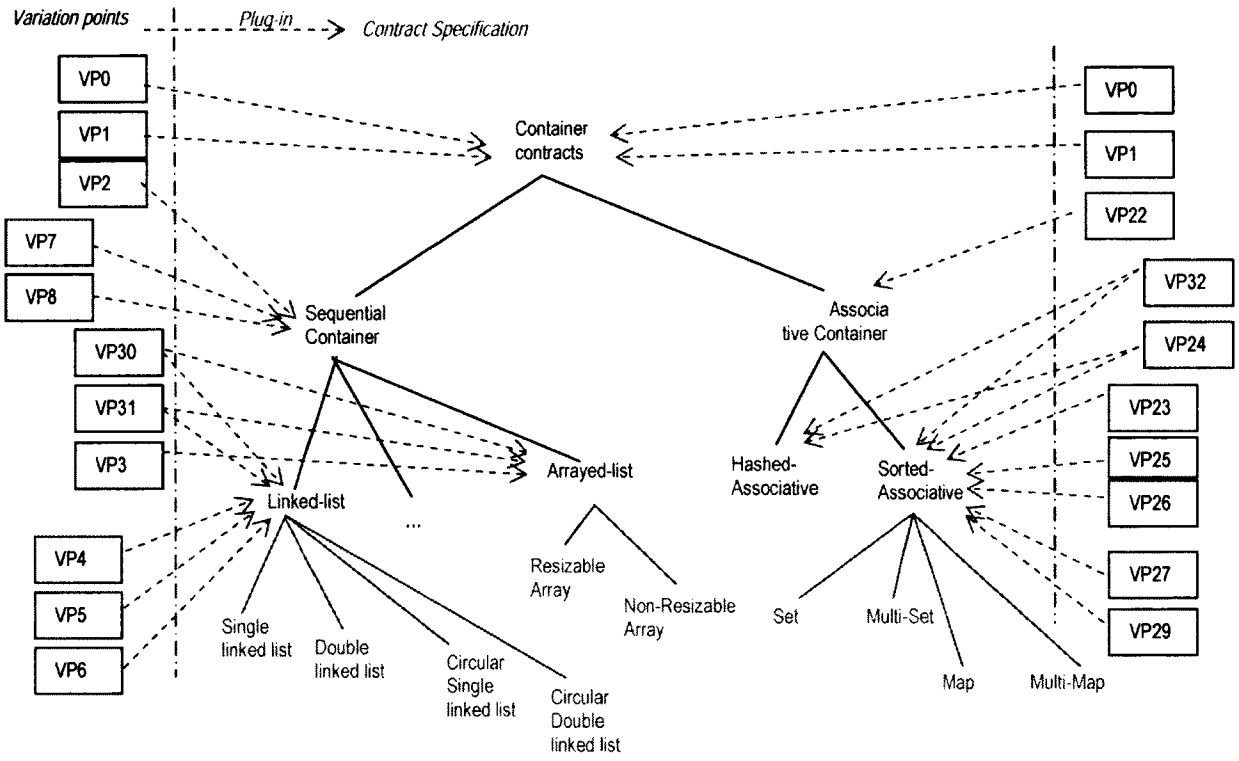

Figure 7.1 - Contract Hierarchies for the Domain of Sequential Containers (left) and the Domain of Associative Containers (right) with their Variabilities and Plug-in points

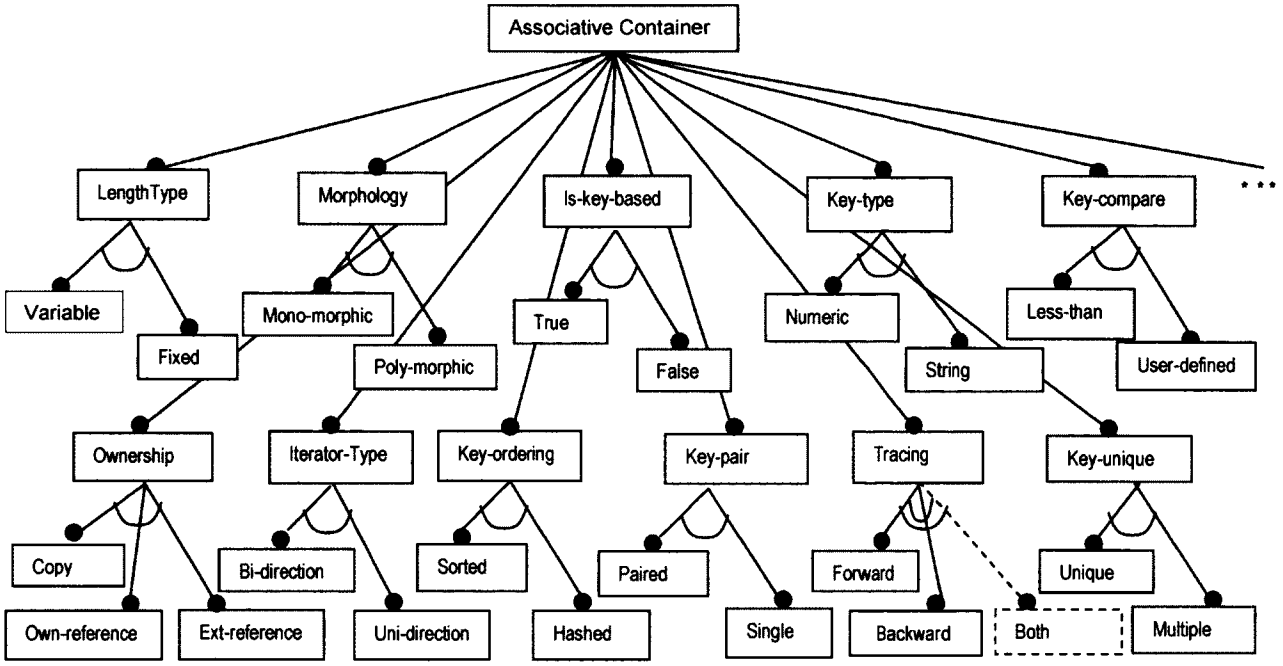

Figure 7.2 - Feature diagram for Associative Containers

VP0 and VP1 are the same as for sequential containers. Here is a short description of the other variation points:

- Variation point VP22 "Morphology" has two variants VP22-1 "Mono-morphic" and VP22-2 "Poly-morphic". It indicates whether the elements of the container share a same type or not. As each element is referred to by its key, thus the 'type of an element' refers to the type of the data (i.e., value) part of each element.

- Variation point VP23 "Ownership" has three variants [CZA99], VP23-1 "Copy", VP32-2 "Owned-reference", and VP32-3 "External-reference". The first variant implies the container keeps copies of the original elements and is responsible to allocate and de-allocate them. The second one denotes a container that keeps references to the original elements and again it is responsible to both allocate and de-allocate them. The

third variant corresponds to a container that keeps references to original elements but it is not responsible to allocate and de-allocate them.

- Variation point VP24 "iterator-type" has two variants VP24-1 "Bi-direction" and VP24-2 "Uni-direction". The first variant means the elements in the associative container can be traversed in both ways forward and backward, while the second variant means the elements can only be traversed in a forward way. In the latter case, an external operator (i.e., one not owned by the container iterator) should be defined to access the previous elements, while in the former case both next and previous elements can be accessed using operators owned by the container iterator. This variation point shows a typical change for a requirement from a common to a variable one, and that how our modeling and generative framework can handle this new variation point and its potential affects on the domain contracts. In our case studies, iterator was a common requirement for the sequential containers (case study I) that has been changed to a variation point for the associative ones (case study II), see the contracts below.

- Variation point VP25 "Key-unique" has two variants VP25-1 "Unique" and VP25-2 "Multiple". The first variant means that each element's key is unique, and the second variant means that each element's key is not necessarily unique (i.e., can be reused by other elements) and thus the container can have multiple keys.

- Variation point VP26 "Key-pair" has two variants VP26-1 "Paired" and VP26-2 "Single". The first variant means that each element's value contains a pair (formed by its key and a data object), and the second variant means that each element's value is the same as its key.

- Variation point VP27 "Key-compare" has two variants VP27-1 "Less-than" and VP27-2 "User-defined". The first variant means that the comparison between two keys is based on less-than operation, which is the default operation in associative containers, and the second variant means that the comparison between two keys is based on a user-defined operation.

- Variation point VP29 "Key-Type" has two variants VP29-1 "Numeric" and VP29-2 "String". The first variant means that the key in each element is a numeric type, and the second one means that the key is a string type.

- Variation point VP32 "Key-ordering" has two variants VP32-1 "Sorted" and VP32-2 "Hashed". The first variant means that the associative container has keys in ascending order, which is the default order and explains why the less-than operator is the default operator for comparing the keys. The second variant means the associative container has no-order on its elements with respect to their keys and that they can be accessed via a hash mechanism.

- Variation point VP34 "Tracing" has two variants VP34-1 "Forward" and VP34-2 "Backward". The first variant indicates that the associative container has a mechanism to store the links among the variabilities in a forward manner – variability in each step can be linked to the relevant variability in the next step in our model-driven approach –, and the second variant means that the associative container has a mechanism to store the links to variabilities in a backward manner – variability in each step can be linked to the relevant variability in the previous step in our model-driven approach. As this variation point has an "OR" relation among its variants, we consider an extra virtual variant for it – in order to have a proper analysis and selection for it – according to rules 1 and 2 in "Variability Relationship Rules" defined in section 3.2 (chapter 3). This extended variant is the combination of the first two variants and is shown with a dashed line in the Associative feature diagram (Figure 7.2).

In order to illustrate the use of these features, a multimap can be specified using the following configuration:

```
<VP0=True, VP1=Variable, VP22=Mono-morphic, VP23=Copy, VP4=Bi-direction,
VP25=Multiple, VP26=Paired, VP27=Less-than, VP29=Numeric, VP32=Sorted,
VP34=Both>
```

The relationships between these features are captured in the following table (Table 7.1), in which the rightmost column provides an explanation of the rule captured:

Table 7.1 - Variability Relationships Table (RT) for Associative Containers

No.	Related VP(s) Var(s)	Related VP & Var Types	Related VP & Var Names	Relation: Rule# Constraint and Actions Syntax	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
2	VP1 VP1-1 VP1-2	Var. point Variant Variant	"LengthType" "Variable" "Fixed"	Rule 1, 2: VP1 ←X VP1-1 VP1 ←X VP1-2	VP1 <depends> VP1-1, VP1-2 Cond = - Action = variant
3	VP22 VP22-1 VP22-2	Var. point Variant Variant	"Morphology" "Mono-morphic" "Poly-morphic"	Rule 1, 2: VP22 ←X VP22-1 VP22 ←X VP22-2	VP22 <depends> VP22-1, VP22-2 Cond = - Action = variant
4	VP23 VP23-1 VP23-2 VP23-3	Var. point Variant Variant Variant	"Ownership" "Copy" "Own-reference" "Ext-reference"	Rule 1, 2: VP23 ←X VP23-1 VP23 ←X VP23-2 VP23 ←X VP23-3	VP23 <depends> VP23-1, VP23-2, VP23-3 Cond = - Action = variant
5	VP24 VP24-1 VP24-2	Var. point Variant Variant	"Iterator-Type" "Bi-direction" "Uni-direction"	Rule 1, 2: VP24 ←X VP24-1 VP24 ←X VP24-2	VP24 <depends> VP24-1, VP24-2 Cond = - Action = variant
6	VP25 VP25-1 VP25-2	Var. point Variant Variant	"Key-unique" "Unique" "Multiple"	Rule 1, 2: VP25 ←X VP25-1 VP25 ←X VP25-2	VP25 <depends> VP25-1, VP25-2 Cond = - Action = variant
7	VP26 VP26-1 VP26-2	Var. point Variant Variant	"Key-pair" "Paired" "Single"	Rule 1, 2: VP26 ←X VP26-1 VP26 ←X VP26-2	VP26 <depends> VP26-1, VP26-2 Cond = - Action = variant
8	VP27 VP27-1 VP27-2	Var. point Variant Variant	"Key-compare" "Less-than" "User-defined"	Rule 1, 2: VP27 ←X VP27-1 VP27 ←X VP27-2	VP27 <depends> VP27-1, VP27-2 Cond = - Action = variant
9	VP29 VP29-1 VP29-2	Var. point Variant Variant	"Key-Type" "Numeric" "String"	Rule 1, 2: VP29 ←X VP29-1 VP29 ←X VP29-2	VP29 <depends> VP29-1, VP29-2 Cond = - Action = variant
10	VP32 VP32-1 VP32-2	Var. point Variant Variant	"Key-ordering" "Sorted" "Hashed"	Rule 1, 2: VP32←X VP32-1 VP33←X VP32-2	VP32 <depends> VP32-1, VP32-2 Cond = - Action = variant
11	VP34 VP34-1 VP34-2 VP34-3	Var. point Variant Variant Variant	"Tracing" "Forward" "Backward" "Both"	Rule 1, 2: VP34←R VP34-1 VP34←R VP34-2 VP34←R VP34-3	VP34<depends>VP34-1, VP34-2, VP34-3 Cond = - Action = variant
12	VP32 VP32-2 VP27 VP27-2	Var. point Variant Var. point Variant	_	Rule 5: If(VP32==VP32-2) IMPLIES... VP27==VP27-2	VP32-2<requires>VP27-2 Cond1="VP27==User-defined" Action= variant1
13	VP25 VP25-1 VP25-2 VP0 VP0-1	Var. point Variant Variant Var. point Variant	_	Rule 3: If(VP25 != null && (VP25== VP25-1 || VP25 == VP25-2)) IMPLIES VP0 == VP0-1	VP25 <requires> VP0-1 Cond1 = "VP0 ==True" Action = variant1

Table 7.1 - Variability Relationships Table (RT) for Associative Containers (cont.)

No.	Related VP(s) Var(s)	Related VP & Var Types	Related VP & Var Names	Relation: Rule# Constraint and Actions Syntax	Relation: <depends>, <requires>,<excludes> Constraints and Actions in Contracts
14	VP22 VP22-2 VP26 VP26-1	Var. point Variant Var. point Variant	_	Rule 5: If(VP22==VP22-2) IMPLIES... VP26==VP26-1	VP22-2<requires>VP26-1 Cond1="VP26==Paired" Action= variant1
15	VP22 VP22-2 VP26 VP26-2	Var. point Variant Var. point Variant	_	Rule 6: If(VP22==VP22-2) IMPLIES... VP26 !=VP26-2	VP22-2<requires>!VP26-2 Implies: Cond1="VP26 !=Single" or Cond2="VP26 == Single" Action= variant2
16	VP26 VP26-1 VP26-2 VP0 VP0-1	Var. point Variant Variant Var. point Variant	_	Rule 3: If(VP26 != null && (VP26== VP26-1 || VP26 == VP26-2)) IMPLIES VP0 == VP0-1	VP26 <requires> VP0-1 Cond1 = "VP0 ==True" Action = variant1
17	VP27 VP27-1 VP27-2 VP0 VP0-1	Var. point Variant Variant Var. point Variant	_	Rule 3: If(VP27 != null && (VP27== VP27-1 || VP27 == VP27-2)) IMPLIES VP0 == VP0-1	VP27 <requires> VP0-1 Cond1 = "VP0 ==True" Action = variant1
18	VP29 VP29-1 VP29-2 VP0 VP0-1	Var. point Variant Variant Var. point Variant	_	Rule 3: If(VP29 != null && (VP29== VP29-1 || VP29 == VP29-2)) IMPLIES VP0 == VP0-1	VP29 <requires> VP0-1 Cond1 = "VP0 ==True" Action = variant1
19	VP32 VP32-1 VP32-2 VP0 VP0-1	Var. point Variant Variant Var. point Variant	_	Rule 3: If(VP32 != null && (VP32== VP32-1 || VP32 == VP32-2)) IMPLIES VP0 == VP0-1	VP32 <requires> VP0-1 Cond1 = "VP0 ==True" Action = variant1

Interestingly, the abstract contained is the same as for sequential containers (as it should be). We give the main contract for the domain of associative containers (with explanations embedded in the examples in the form of comments):

```
Contract Iterator
    //same as for previous case study
 // ------------------------------------------------------------------------
Contract Item
    {
       Observability Key getKey();//returns key of item
       Observability T getData(); //returns data of item

       Responsibility new()
       {
          Post(getKey() > 0);
       }

       Responsibility finalize()
       {
          Pre(getKey() > 0);
       }
    }// End Contract
// ------------------------------------------------------------------------
    MainContract Container extends ContainerBase<tItem>
    {
          //defining variables
       Scalar key_type keyNext;
       Scalar key_type keyPrev;
       Scalar key_type k;
       Scalar data_type t;
       Scalar value_type x;
       Scalar tIterator tobeNext;
       Scalar tIterator tobePrev;
       Scalar tIterator new_pos;
       Scalar Integer old_count;
       Scalar Integer number_of_items;
       Scalar Boolean new_insert;

       // The size(), max_size(), and IsEmpty() observabilities
       // will be copied to this contract
       Observability tIterator Begin();//returns pointer to first item
       Observability tIterator End();//returns pointer to last item
       Observability value_type itemAt(tIterator pos);//value at pos
       Observability tIterator findNext(tIterator pos);//iterate forward
       //plugin VP24();

       Observability tIterator find(key_type k);//pointer to item with k
       Observability Integer count(key_type k);//counts numbers of k
       // return number of elements between i and j
       Observability Integer count_range(tIterator i, tIterator j);
       // returns true if k1 and k2 obey the ordering function
       Observability Boolean key_comp(key_type k1,key_type k2);
       //keys are inserted based on Hash function or not
       //plugin VP32();
       //Observability tIterator hash_ret(key_type k);

       //tracing links for variability
       //plugin VP34();

       Observability tIterator find_prev(tIterator pos)
       {
          //plugin VP24();
       }

       Observability Boolean comp(key_type k1, key_type k2)
       {
           //comapres the keys of two elements
```

```
            //plugin VP27();
        }

        Observability data_type find_data(value_type x)
        {
            //returns the data part of an item
            //plugin VP26();
        }

        Observability key_type find_key(value_type x)
        {
            //returns the key part of an item
            //plugin VP26();
        }
```

/* The input is the return value of the *insert()* responsibility that may have a paired structure or not. In either case, the return value should contain an iterator pointing to the inserted element. This observability extracts and returns that iterator as output.
*/

```
        Observability tIterator find_ret(ret_type p)
        {
            //returns the iterator part of a ret_type
            //plugin VP25();
        }

        Observability Boolean check_data_type(value_type x)
        {
            //check if the new item has the right type
            //plugin VP22();
        }
```

/* has two inputs: the first input is an iterator to the inserted element, and the second input is the inserted key. First it will find the iterators to the next and the previous elements through *findNext()* and *findPrev()* observabilities. Then it will find the keys of the next and the previous elements through *itemAt()* and *find_key()* observabilities. And finally it will compare the inserted key with the keys of the next and the previous elements through *comp()* observability. If all the three keys, the previous, inserted and the next keys, are in the correct order it will return true, false otherwise.
*/

```
        Observability Boolean check_inserted_key
            (tIterator inserted, key_type inserted_key)
        {
            Scalar Boolean result=false;
            tobeNext = findNext(inserted);
            tobePrev = find_prev(inserted);
            x = itemAt(tobeNext);
            keyNext = find_key(x);
            x = itemAt(tobePrev);
            keyPrev = find_key(x);
            result=(comp(inserted_key, keyNext) == true
                && comp(keyPrev, inserted_key) == true);
            value = result;
        }

        //irrelevant contract elements are omitted here
```

/*Inserting an item with a paired return-type: second part is true, it is inserted, first is an iterator to the inserted position.

The return value is a paired structure if the container has unique keys, and is an iterator to the inserted item if the container has multiple keys. In the former case the first part of the pair is an iterator to the inserted item and the second part of it is a Boolean value that indicates if the insertion takes place or not. We have defined observability *find_ret()* to extract these two parts from the return value. Before the IUT execution, it checks if the input item x - the item to be inserted - has a correct data type through a pre-condition and *check_data_type()*.The data-type is correct if it is the same for all the elements in a mono-morphic container and it is always correct for a poly-morphic container. After the IUT insertion, it finds the position of the new inserted element through *find_ret()* and saves it in variable *new_pos*. It then checks if the key in the *new_pos* is in the correct order through a post-condition and *check_inserted_key()*. Then it checks if the inserted item at the *new_pos* has a data-part equal to the data-part of the input item through a post- condition, *itemAt()* and *find_data()*. And finally it checks if the number of the inserted key has been increased by one or not, through a post- condition and *count()*. For this reason, the number of the inserted key was saved in variable *old_count* before the insertion.

Other insertions are similar.
*/

```
Responsibility ret_type insert(value_type x)
   extends  GenericInsertion()
{
   k = find_key(x);
   old_count = count(k);
   //plugin VP25();
   //Pre(old_count == 0);
   Pre(check_data_type(x) == true);
   Execute();
   new_pos = find_ret(value);
   Post(check_inserted_key(new_pos, k) == true);//ordered keys
   Post(find_data(itemAt(new_pos)) == find_data(x));
   Post(count(k) == old_count + 1);
}

//Inserting an item with iterator return-type to inserted position
Responsibility tIterator insertAt(tIterator pos, value_type x)
   extends GenericInsertion()
{
   k == find_key(x);
   old_count = count(k);
   //plugin VP25();
   //Pre(old_count == 0);
   Pre(check_data_type(x) == true);
   Execute();
   new_pos = value;
   Post(check_inserted_key(new_pos, k) == true);
   Post(find_data(itemAt(new_pos)) == find_data(x));
   Post(count(k) == old_count + 1);
}

//Inserting a range of items between i and j
Responsibility Void insertRange(tIterator i, tIterator j)
{
   number_of_items = count_range(i, j);
   Execute();
   loop(0 to number_of_items)
   {
      x = itemAt(i);
      k = find_key(x);
      Post(find(k) not= End());
      i = findNext(i);
```

```
        }
    }

    //Deleting item with key: all deletions are straightforward.
    Responsibility Integer delete(key_type k) extends GenericDeletion()
    {
        old_count = count(k);
        //plugin VP25();
        //Pre(old_count == 1);
        Execute();
        number_of_items = value;
        Post(number_of_items == old_count);
        Post(count(k) == 0);
        Post(size() == old_Size - number_of_items);
    }

    //Deleting item at position
    Responsibility Void deleteAt(tIterator pos) extends GenericDeletion()
    {
        x = itemAt(pos);
        k = find_key(x);
        old_count = count(k);
        //plugin VP25();
        //Pre(old_count == 1);
        Execute();
        Post(count(k) == old_count - 1);
        Post(size() == old_Size - 1);
        //plugin VP23();
        //Post(pos == null);
    }

    //Deleting items between i and j
    Responsibility Void deleteRange(tIterator i, tIterator j)
    {
        number_of_items = count_range(i, j);
        tobeNext = findNext(j);
        tobePrev = find_prev(i);
        Execute();
        x = itemAt(tobeNext);
        keyNext = find_key(x);
        x = itemAt(tobePrev);
        keyPrev = find_key(x);
        Post(comp(keyPrev, keyNext) == true);
        Post(size() == old_Size - number_of_items);
    }

    //irrelevant contract elements are omitted here

// exports section: to connect the container's Item type
    Exports
    {
        Type tItem conforms Item
        {
            not context;
            not derived context;
        }
        Type tIterator conforms Iterator;

        //plugin VP25();
        //Type ret_type conforms pair<tIterator, Boolean>;

        //plugin VP0();
        //Type key_type conforms Key;
```

```
            //plugin VP26();
            //Type data_type conforms T;
            //Type value_type conforms pair<key_type, data_type>;

            //plugin VP29();
            // Type Key conforms Integer;
        }
    }// End Contract
```

The variability contracts, which proceed from the table (Table 7.1) given earlier, follow:

```
Variability Repository Domain: Associative Containers
{
  Variation VP0 <Is-key-based> [1..1] out of 2
  {
    case "True":
     plug-in: VP0-1 //Item is associated with a key
     Refine-a: Exports
     Type key_type
     {
           Type key_type conforms Key;
     }
    case "False":
     plug-in: VP0-2 //Item is not associated with a key
     Refine-a: Exports
     Type key_type
     { }
  }

  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());
     }
  }

  Variation VP22 <Morphology> [1..1] out of 2
  {
    case "Mono-morphic":
     Plug-in: VP22-1 //Elements have same data type
     Refine-a: Observability
     Boolean check_data_type(value_type x)
     {//a dynamic type checking to ensure the data is of data_type
        value = (typeid(find_data(x)) == typeid(data_type));
     }
    case "Poly-morphic":
     Plug-in: VP22-2 //Elements have different data types
     cond1: ("VP26 == Paired and VP0 == True")
     cond2: ("VP26 == Single and VP0 == True")//mutually exclusive with paired
     Refine-a: Observability
```

```
    Boolean check_data_type(value_type x)
    {//no dynamic type checking is needed
        #IF (cond1)
            value = true;
        #ENDIF
        #IF (cond2)
            value = false;
        #ENDIF
    }
}

Variation VP23 <Ownership> [1..1] out of 3
{
  case "Copy":
    Plug-in: VP23-1 //container owns copy of items
    Refine-a: Responsibility
    Void deleteAt(tIterator pos)
    {}
  case "Owned-reference":
    Plug-in: VP23-2 //container owns references to items
    Refine-a: Responsibility
    Void deleteAt(tIterator pos)
    {
        Post(pos == null);
    }
  case "External-reference":
    Plug-in: VP23-3 //container doesn't own references to items
    Refine-a: Responsibility
    Void deleteAt(tIterator pos)
    {}
}

Variation VP24 <Iterator-type> [1..1] out of 2
{
  case "Bi-direction":
    Plug-in: VP24-1 //Iterate container is bi-direction
     Refine-a: Declaration
    tIterator find_prev
    {
        Observability tIterator findPrev(tIterator pos);
    }
    Refine-a: Observability
    tIterator find_prev(tIterator pos)
    {
        value = findPrev(pos);
    }

  case "Uni-direction":
    Plug-in: VP24-2 //Iterate container is uni-direction
    Refine-a: Declaration
    tIterator find_prev
    {
        Observability tIterator Previous(tIterator pos);
    }
    Refine-a: Observability
        tIterator find_prev(tIterator pos)
    {
        value = Previous(pos);
    }
}


Variation VP25 <Key-unique> [1..1] out of 2
```

```
{
  case "Unique":
   Plug-in: VP25-1 //Container has unique keys
   cond1: ("VP0 == True")
   Refine-a: Exports
   Type ret_type
   {   #IF (cond1)
          Type ret_type conforms pair<tIterator, Boolean>;
       #ENDIF
   }
   Refine-a: Observability
   tIterator find_ret(ret_type p)
   {   #IF (cond1)
          List l1 = p;
          new_insert = l1.indexOf(2);//p.second or boolean
          value = l1.indexOf(1);//p.first or iterator
        #ENDIF
   }
   Refine-a: Responsibility
   ret_type insert(value_type x)
   {   #IF (cond1)
          Pre(old_count == 0);
       #ENDIF
   }
   Refine-a: Responsibility
   tIterator insertAt(tIterator pos, value_type x)
   {  #IF (cond1)
          Pre(old_count == 0);
      #ENDIF
   }
   Refine-a: Responsibility
   Integer delete(key_type k)
   {  #IF (cond1)
          Pre(old_count == 1);
      #ENDIF
   }
   Refine-a: Responsibility
   Void deleteAt(tIterator pos)
   {  #IF (cond1)
          Pre(old_count == 1);
      #ENDIF
   }

  case "Multiple":
   Plug-in: VP25-2 //Container has equal keys
   cond1: ("VP0 == True")
   Refine-a: Exports
   Type ret_type
   {  #IF (cond1)
          Type ret_type conforms Iterator;
      #ENDIF
   }
   Refine-a: Observability
   tIterator find_ret(ret_type p)
   {   #IF (cond1)
          new_insert = true;
          value = p; //p is iterator
       #ENDIF
   }
   Refine-a: Responsibility
   ret_type insert(value_type x)
   {
   }
```

```
    Refine-a: Responsibility
    tIterator insertAt(tIterator pos, value_type x)
    {
    }
    Refine-a: Responsibility
    Integer delete(key_type k)
    {  #IF (cond1)
           Pre(old_count not= 0);
       #ENDIF
    }
    Refine-a: Responsibility
    Void deleteAt(tIterator pos)
    {   #IF (cond1)
           Pre(old_count not= 0);
        #ENDIF
    }
}

Variation VP26 <Key-pair> [1..1] out of 2
{
  case "Paired":
    Plug-in: VP26-1 //Each element has a paired key
    Cond1: ("VP0 == True")
    Refine-a: Exports
    Type value_type
    {  #IF (cond1)
           Type data_type conforms T;
           Type value_type conforms pair<key_type, data_type>;
       #ENDIF
    }
    Refine-a: Observability
    data_type find_data(value_type x)
    {  #IF (cond1)
           Scalar data_type t1;
           t1 = x.getData();//x.second or x.data
           value = t1;
       #ENDIF
    }
    Refine-a: Observability
    key_type find_key(value_type x)
    {  #IF (cond1)
           Scalar key_type k1;
           k1 = x.getKey();//x.first or x.key
           value = k1;
       #ENDIF
    }
  case "Single":
    Plug-in: VP26-2 //Each element is the key itself
    Cond1: ("VP0 == True")
    Refine-a: Exports
    Type value_type
    {  #IF (cond1)
           Type data_type conforms Key;
           Type value_type conforms Key;
       #ENDIF
    }
    Refine-a: Observability
    data_type find_data(value_type x)
    {  #IF (cond1)
           value = x;
       #ENDIF
    }
    Refine-a: Observability
```

```
    key_type find_key(value_type x)
    {  #IF (cond1)
           value = x;
       #ENDIF
    }
}

Variation VP27 <Key-compare> [1..1] out of 2
{
  case "Less-than":
    Plug-in: VP27-1 //Compare keys with less than operator
    Cond1: ("VP0 == True")
    Refine-a: Observability
    Boolean comp(key-type k1, key-type k2)
    {   #IF (cond1)
           value = (k1 < k2);
        #ENDIF
    }
  case "User-defined":
    Plug-in: VP27-2 //Compare keys with user-defined operator
    Cond1: ("VP0 == True")
    Refine-a: Observability
    Boolean comp(key-type k1, key-type k2)
    {   #IF (cond1)
           value = key_comp(k1, k2);
        #ENDIF
    }
}

Variation VP29 <Key-Type> [1..1] out of 2
{
  case "Numeric":
    Plug-in: VP29-1 //Key type is numeric
    Cond1: ("VP0 == True")
    Refine-a: Exports
    Type Key
    {
      #IF (cond1)
           Type Key conforms Integer;
      #ENDIF
    }
  case "String":
    Plug-in: VP29-2 //Key type is alphanumeric
    Cond1: ("VP0 == True")
    Refine-a: Exports
    Type Key
    {
      #IF (cond1)
           Type Key conforms String;
      #ENDIF
    }
}

Variation VP32 <Key-ordering> [1..1] out of 2
{
  case "Sorted":
    Plug-in: VP32-1 //keys are inserted and sorted based on some total order
    cond1: ("VP0 == True")
    Refine-a: Declaration
    tIterator hash_ret
```

```
    {}
  case "Hashed":
    Plug-in: VP32-2 //keys are inserted and sorted based on a hash result
    cond1: ("VP27 == User-defined and VP0 == True")
    Refine-a: Declaration
    tIterator hash_ret
    {     #IF (cond1)
            Observability tIterator hash_ret(key_type k);
          #ENDIF
    }
}

Variation VP34 <Tracing> [1..1] out of 3
{
  case "Forward":
    Plug-in: VP34-1 //create forward links from RT table
    Refine-a: Declaration
    Void Dump-Tracing-links
     {
           Observability Void Dump-Forward-links();
     }
  case "Backward":
    Plug-in: VP34-2 //create backward links from RT table
    Refine-a: Declaration
    Void Dump-Tracing-links
    {
           Observability Void Dump-Backward-links();
    }
  case "Both":
    Plug-in: VP34-3 //create forward & backward links from RT
    Refine-a: Declaration
    Void Dump-Tracing-links
    {
           Observability Void Dump-Forward-links();
           Observability Void Dump-Backward-links();
    }
}

}//End of Contracts
```

The XML counterparts of the domain and variability contracts are of little interest and thus are omitted. The generative process was tested with several configurations including sets, multi-sets, maps, and multi-maps.

8 Conclusion

8.1 Recapitulation

As previously mentioned, this dissertation can be viewed as a study of how feasible it is to implement an XML template-based generative approach for a contract-based requirements specification language for model-based testing. Our choice of ACL [ARN09-a] (and of its supporting tool VF [ARN09-b]) has been discussed in the first chapter of this dissertation. The main contribution of this work is a domain- and language-independent generative process we propose for generating 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). Our work not only illustrates the power and generality of the template-based approach to generation (advocated by Cleaveland [CLEA88] and [CLEA01]) but also, the importance of traceability in order to avoid the 'semantic gaps' present in some of the existing generative approaches. Furthermore, the two case studies we have developed emphasize another distinguishing characteristic of our work: the generated artifact is not only traceable 'backward' to a feature diagram and a feature grammar, but also 'forward' to actual code. This allows the generated artifact to be not only compiled but also run against an implementation under test, *de facto* offering a strong validation approach for the whole generative process.

- Clearly, the automation of the generative process we propose has highest priority at this point in time (which explains why another member of our research group has already started working on it). A fully automated generative process will not only implement the transformations we have put forth throughout this dissertation, it will be able to verify the outcome of these transformations via the implementation and use of the traceability links we suggest (section 6.2.1).

- Once other features of ACL stabilize, in particular scenarios, it will be highly desirable to update our generative process to include them. Most importantly, tackling scenarios (through a replace refinement "refine-r"), metrics and other ACL features will not change the fundamental approach, that is, the use of XML-based selection templates. This is a crucial point that reinforces Cleaveland's [CLEA01] claims with respect to the numerous benefits of such an approach.

- Radonjic ([RAD-11a], [RAD11-b]) is looking into the design of static and dynamic ACL checks that will pertain to design patterns [GAMM95]. He is defining *rules* to determine if a given design pattern is being used according to its intended use. These *rules* will be using both static and dynamic checks in ACL contracts that can be executed against a candidate IUT to ensure the correct use of the design patterns implemented in that IUT. Using our variability modeling generative framework, we intend to extend his work in order to verify the correct use of design patterns for a family of systems.

- Non-functional requirements (NFRs) define criteria that can be used to evaluate *how* a system satisfies its requirements (as opposed to *what* requirements it satisfies). For example, reliability, maintainability, and performance are non-functional requirements that can illustrate how a system operates in terms of speed, timing, memory usage, coupling between procedures, etc. NFRs can be represented as soft-goals using the GRL notation [MYLO01], [CHU00], and their

relationships with functional goals can be represented using GRL interdependency graphs. The measurement of metrics is relevant to determining the (positive or negative) influence a functional unit contributes to the satisfaction of a soft-goal. We plan to investigate how the soft-goals within a contract-based domain model can be affected by the variable requirements of the domain. This can be done by integrating our work on vriability with the GRL interdependency graph, so that each variant can affect (either positively or negatively) some specific relevant NFRs of the domain.

- We believe it would be extremely useful to eventually develop a catalogue of contracts for frequently used family of systems, particularly for widely used libraries. Such a catalogue would promote reuse and ease the task of selecting from a set of potential implementations by allowing its user to run the generated catalogue member (according to a desired configuration list) to determine if a candidate implementation conforms or not to the required functionality for that member.

- In a number of recent works: [LAH07], [MORI07], [MORI08] the authors introduced a generic Aspect Oriented Modeling (AOM) Framework weaving with variability in order to create complex systems. In fact reusability was very limited in current AOM approaches, as they only described one possible variant for an aspect and offered only one way to weave that variant into the target model. Later, the authors introduced two protocols of 'matching' and 'adaptation' of variability into the aspects to increase the reuse. Comparing to our work, in AOM approaches there is still a big gap between the requirements' model and the IUTs, and between weaving variability into the target model and configuring variability for that model. In our future work we would like to investigate more on introducing variability mechanisms into AOM approaches and improving weaving mechanisms to make aspects more configurable and reusable.

9 References

[ALE01] Alexandrescu, A.: "Modern C++ Design: Generic Programming and Design Patterns Applied", Addison-Wesley, ISBN 0-201-70431-5, Feb 13, 2001.

[ANT04] Antkiewicz, M., Czarnecki, K.: "FeaturePlugin: Feature modeling plug-in for Eclipse", In Proceedings of the ACM European conference on object-oriented programming systems, languages, and applications, Conference (OOPSLA'04), Eclipse Technology eXchange (ETX) Workshop, pp. 67-72, 2004.

[ARN09-a] Arnold, D.: "Supporting Generative Contracts in .Net", Doctoral dissertation, School of Computer Science, Carleton University, April 2009.

[ARN09-b] Arnold, D.: "Validation Framework and Another Contract Language", http://vf.davearnold.ca/, last accessed in October 2012.

[ARN10] Arnold, D., Corriveau, J.-P., Shi, W.: "Validation against Actual Behavior: Still a Challenge for Testing Tools", International Conference on Software Engineering Research and Practice (SERP 2010), Las Vegas, USA, pp.212-218, July 2010.

[ATL] A Model Transformation Language, http://www.eclipse.org/atl/, last accessed in October 2012.

[ATLAS] Atlantic Modeling, AtlanMod, http://www.emn.fr/z-info/atlanmod/index.php/Main_Page, last accessed in August 2012.

[BAC03] Bachmann, F., Goedicke, M., do Prado Leite, J. C. S., Nord, R. L., Pohl, K., Ramesh, B., Vilbig, A.: "A metamodel for representing variability in product family development", In PFE'03: Proc. of the 5th International Workshop on Software Product-Family Engineering, pp. 66-80, 2003.

[BAS07] Bassett, P. G.: "The Case for Frame-Based Software Engineering", IEEE Software, vol. 24, no. 4, pp. 90-99, July 2007.

[BAS97] Bassett, P.: "Framing software reuse - lessons from real world", Yourdon Press, Prentice Hall, Englewood Cliffs, ISBN-10: 013327859X, 1997.

[BASH08] Bashardoust-Tajali, S., Corriveau, J. P.: "On Extracting Tests from a Testable Model in the Context of Domain Engineering", Proceedings of ICECCS08, IEEE International Conference on Engineering Complex Computer Systems, Belfast, pp. 98-107, April 2008.

[BASH11] Bashardoust, S., Radonjic, V.D., Corriveau, J. P.: "Challenges of Variability in Model-Driven and Transformational Approaches: A Systematic Survey", IEEE/IFIP Workshop on Variability in Software Architecture, WICSA2011, Boulder, Colorado, USA, pp. 294-301, 2011.

[BASI05] Basit, H., Rajapakse, D. C., Jarzabek, S.: "Beyond Templates: A Study of Clones in the STL and some General Implications", In Proceeding of International Conference in Software Engeering (ICSE 05), ACM Press, pp. 451-459, 2005.

[BAT03] Batory, D., Sarvela, J.N., Rauschmayer, A.: "Scaling Step-Wise Refinement", In: Proc. Int. Conf. on Software Engineering, ICSE 2003, Portland, Oregon, pp. 187–197, May 2003.

[BAT92] Batory, D., O'Malley, S.: "The Design and Implementation of Hierarchical Software Systems with Reusable Components", ACM Trans. on Software Engineering and Methodology, vol. 1, no. 4, pp. 355–398, 1992.

[BAY06] Bayer, J., Gerard, S., Haugen, Ø., Mansell, J., Møller-Pedersen, B., Oldevik, J., Tessier, P., Thibault, J.-P., Widen, T.: "Consolidated Product Line Variability Modeling", In: Käköla, T., Duenas, J.C. (eds.) Software Product Lines - Research Issues in Engineering and Management, Springer, Heidelberg, ISBN: 3-540-33252-9, pp. 195–241, 2006.

[BER05] Berg, K., Bishop, J., Muthig, D.: "Tracing software product line variability: from problem to solution space", In SAICSIT'05: Proc. of the 2005 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries, Republic of South Africa (SAICSIT), pp. 182-191, 2005.

[BERK03] Berkenkotter, K.: "Using UML 2.0 in Real-Time Development: A Critical Review", In International Workshop On Specification and Validation of UML Models For Real Time and Embedded Systems (SVERTS 2003), San Francisco, USA, pp. 41–54, October 2003.

[BERT07] Bertolino, A.: "Software Testing Research: Achievements, Challenges and Dreams", In IEEE – Future of Software Engineering (FOSE '07), Minneapolis, pp. 85-103, May 2007.

[BEZ03-a] Bézivin, J., Dupé, G., Jouault, F., Pitette, G., Rougui, EJ: "First experiments with the ATL model transformation language: Transforming XSLT into XQuery", In Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'03) Workshop, California, 2003.

[BEZ03-b] Bézivin, J., Farcet, N., Jézéquel, J.-M., Langlois, B., Pollet, D. : "Reflective model driven engineering", In G. Booch P. Stevens, J. Whittle, editor, Proceedings of UML 2003, vol. 2863 of LNCS, San Francisco, pp. 175-189, October 2003.

[BEZ05] Bezivin, J., Heckel, R., editors: "Language Engineering for Model-Driven Software Development", In Proceedings of Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI'05), Schloss Dagstuhl, Germany, pp. 208- 219, 2005.

[BIN99] Binder, R.: "Testing Object-Oriented Systems", Addison-Wesley Professional, 1999.

[BLOU07] Blouin, A., Beaudoux, O.: "Mapping Paradigm for Document Transformation", DocEng '07: Proceedings of the 2007 ACM symposium on Document engineering, pp. 219-221, 2007.

[BOSCH00] Bosch, J.: "Design and Use of Software Architectures -Adopting and Evolving CI Product Line Approach ", Addison-Wesley, ISBN 020 167494-7, 2000.

[BOSCH02] Bosch, J., Florijn, G., Greefhorst, D., Kuusela, J., Obbink, H., Pohl, K.: "Variability Issues in Software Product Lines" In: Proceedings of the 4th International Workshop on Product Family Engineering (PFE-4), Bilbao, Spain, October 3–*5, 2001*, Springer, Berlin Heidelberg New York, LNCS 2290, pp. 13–21, 2002.

[BUD03] Budinsky F., Steinberg D., Merks Ed.: "Eclipse Modeling Framework: A developer's guide", Addison-Wesley, ISBN: 0-13-142542-0, 2003.

[BUH03] Bühne, S.; Halmans, G., Pohl, K.: "Modeling Dependencies between Variation Points in Use Case Diagrams", In: Salinesi, C.; Regnell, B.; Kamsties, E. (Eds.) Proceedings of 9th Intl. Workshop on Requirements Engineering – Foundation for Software Quality (REFSQ '03), Universitätsbibliothek Essen, Klagenfurt/Velden, Österreich, pp. 59-70, June 2003.

[BUH04] Bühne, S., Lauenroth, K., Pohl, K.: "Why is it not Sufficient to Model Requirements Variability with Feature Models?", International Workshop on Automotive Requirements Engineering (AuRE 2004) co-located at RE04, Nazan University, Nagoya, Japan, 2004.

[BUH05] Bühne, S., Lauenroth, K., Pohl, K.: "Modelling Requirements Variability across Product Lines", 13th IEEE International Requirements Engineering Conference, Paris, France, pp. 41-52, 2005.

[CHU00] Chung, L. K., Nixon, B. A., Yu, E., Mylopoulos, J.: "Non-Functional Requirements in Software Engineering", Kluwer Publishing, 2000.

[Claußθ1] Clauß., M., Muller U., Franczyk B.: "Modeling Variability with UML", In Proc. of Net.ObjectDays 2001, Young Researchers Workshop on Generative and Component-Based Software Engineering, Erfurt, Germany, pp. 226–230, September 2001.

[CLEA01] Cleaveland, C.: "Program Generators with XML and Java", Prentice-Hall, Upper Saddle River, NJ, ISBN-10: 0130258784, Jan. 2001.

[CLEA88] Cleaveland, C.: "Building Application Generators", In IEEE Software, no. 4, vol. 9, pp. 25-33, July 1988, http://craigc.com/pg/index.html, last accessed in October 2012.

[CODW] CodeWorker, http://codeworker.free.fr/, last accessed in August 2012.

[COP98] Coplien, J., Hoffman, D., Weiss, D.: "Commonality and Variability in Software Engineering", Bell Labs, IEEE Software, pp.37-45, Dec. 1998.

[COR96] Corriveau, J. P.: "Traceability Process for Large OO Projects", In IEEE Computer, no. 9, vol. 29, pp. 63-68, Sep 1996.

[COR07-a] Corriveau, J. P.: "Testable Refinable Requirements for Offshore Outsourcing", SEAFOOD: First International Conference on Software Engineering Approaches For Offshore and Outsourced Development, Zurich, Feb. 2007.

[COR07-b] Corriveau, J. P., Bashardoust-Tajali, S., "Generative Hierarchical Contracts for Conformance Testing of Sequential Containers", Proceedings of the 25th conference on IASTED International Conference on Software Engineering, Innsbruck, Austria, pp. 279-284, February 2007.

[COR11-a] Corriveau, J. P., Shi, W.: "Generating Verifiable Test Scenarios", International Conference on Software Engineering Research and Practice (SERP 2011), Las Vegas, USA, vol. 2, pp.315-321, July 2011.

[COR11-b] Corriveau, J. P., Bashardoust-Tajali, S., Radonjic, V. D.: "Requirements Verification in the Presence of Variability – On the feasibility of a Model-Driven Approach", Workshop on Model-Driven Requirements Engineering(MoDRE), RE 2011, Trento, Italy, pp. 74-78, 2011.

[CZA99] Czamecki K., Eisenecker, U.: "Synthesizing Objects", In Proceedings of the 6th European Conference on Object-oriented Programming (ECOOP'99), LNCS, Springer-Verlag, pp. 18-42, 1999.

[CZA00] Czarnecki, K., Eisenecker, U.W.: "Generative Programming: Methods, Tools, and Applications", Addison-wesley, Boston, MA, 2000.

[CZA02] 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.

[CZA03] Czarnecki, K, Helsen, S.: "Classification of Model Transformation Approaches", In: Proceedings of the ACM European conference on object-oriented programming systems, languages, and applications, Conference (OOPSLA'03), Workshop on the Generative Techniques in the Context of Model-Driven Architecture, Anaheim, CA, USA, 2003.

[CZA04-a] Czarnecki, K.: "Overview of Generative Software Development", In: Proceedings of the European Commission and US National Science Foundation Strategic Research Workshop on Unconventional Programming Paradigms (UPP04), September 15–17, Mont Saint-Michel, France, (Lecture Notes in Computer Science vol. 3566), pp. 326-341, 2004.

[CZA04-b] Czarnecki, K., Helsen, S., Eisenecker, U. W.: "Staged configuration using feature models", In Proceedings of the R. L. Nord (ed.), Software Product Lines: Third International Conference, SPLC 2004, Boston, MA, USA, pp. 266-283, 2004.

[CZA05-a] Czarnecki, K., Helsen, S., Eisenecker, U.: "Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models", Software Process: Improvement and Practice, vol. 10, no. 2, pp. 143–169, April/June 2005.

[CZA05-b] Czarnecki, K., Hwan, C., Kim, P.: "Cardinality-Based Feature Modeling and Constraints: A Progress Report", In Proceedings of the ACM European conference on object-oriented programming systems, languages, and applications Conference (OOPSLA'05), Workshop on Software Factories, San Diego, California, USA, October 2005.

[CZA05-c] Czarnecki, K., Antkiewicz, M.: "Mapping Features to Models: A Template Approach Based on Superimposed Variants", In Robert Gluck and Michael R. Lowry, editors, In Proceedings of Generative Programming and Component Engineering, 4th International Conference, GPCE 2005, LNCS 3676, Tallinn, Estonia, pp. 422–437, October 2005.

[CZA05-d] Czarnecki, K., Helsen, S., Eisenecker, U.: "Formalizing cardinality-based feature models and their specialization", Software Process Improvement and Practice, pp. 7–29, 2005.

[CZA05-e] K. Czarnecki, K., M. Antkiewicz, M., C.H.P. Kim, C.H.P., S. Lau, S., and K. Pietroszek, K.: "Model-Driven Software Product Lines", In Proceedings of the ACM European conference on object-oriented programming systems, languages, and applications Conference (OOPSLA'05), San Diego, California, USA, October 2005.

[CZA06] Czarnecki, K., Pietroszek, K.: "Verifying feature-based model templates against well-formedness OCL constraints", In Proceedings of the 5th international conference on Generative programming and component engineering, Portland, Oregon, USA, pp. 221-220, 2006.

[deMI05] deMiguel, M., Exertier, D., Salicki, S. : "Specification of model transformations based on meta-templates", In Jean Bezivin and Robert France, editors, Workshop in Software Model Engineering (WiSME) – Satelite event of the Fifth International Conference on the Unified Modeling Language, p. 204, 2002.

[DJE06] Djebbi, O., Salinesi, C.: "Criteria for Comparing Requirements Variability Modeling Notations for Product Lines", workshop CERE in RE'06, USA, pp. 20-35, September 2006.

[DJE07-a] Djebbi, O., Salinesi, C.: "RED-PL, a method for deriving product requirements from a product line requirements model", in: 19th International Conference on Advanced Information System Engeering (CAiSE'07), Trondheim, Norway, pp. 279-293, 2007.

[DJE07-b] Djebbi, O., Salinesi, C., Diaz, D.: "Deriving product line requirements: the RED-PL guidance approach", in: Asia–Pacific Software Eng. Conf. (APSEC 2007), Nagoya, Japan, pp. 494-501, 2007.

[DJE07-c] Djebbi, O., Salinesi, C., Fanmuy, G.: "Industry survey of product lines management tools: requirements, qualities and open issues", in: 15th IEEE Intl. Requirements Eng. Conf. (RE '07), Delhi, India, pp. 301-306, 2007.

[DOM02] Dominguez, E., Rubio, A., Zapata, M.: "Mapping models between different modeling languages", Workshop on Integration and Transformation of UML models (WITUML), In Proceedings of European Conference on Object-oriented Programming (ECOOP' 2002), pp. 18-22, 2002.

[DOORS] http://www.telelogic.com/doors, last accessed in August 2012.

[DTD] Data Type Definition, http://www.w3.org/TR/REC-xml/#dt-doctype, last accessed in October 2012.

[DUD03-a] Duddy, K., Gerber, A., Raymond, K.: "Eclipse Modelling Framework (EMF) - import/export from MOF / JMI", Cooperative Research Centre for Enterprise Distributed Systems (DSTC) Status Report, Pegamento Project, University of Queensland, Australia, May 2003.

[DUD03-b] Duddy, K., Gerber, A., Lawley, M., Raymond, K., Steel, J.: "Model transformation: A declarative, reusable patterns approach", In: Proc. 7th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2003, Brisbane, Australia, pp. 174-195, 2003.

[DUD04] Duddy, K., Gerber, A., Lawley, M., Raymond, K., Steel, J.: "Declarative transformation for object-oriented models", In van Bommel, P., ed.: Transformation of Knowledge, Information, and Data: Theory and Applications, Idea Group Publishing, 2004.

[ECL] Eclipse Software, http://eclipse.org/, last accessed in October 2012.

[EJBX] XML-based code generator, Houston Technology Group, http://www.theserverside.com/news/thread.tss?thread_id=8429, last accessed in August 2012.

[EMF] Eclipse Modeling Framework, http://www.eclipse.org/modeling/emf/?project=emf, last accessed in October 2012.

[EMR03] Emrich, M.: "Generative Programming Using Frame Technology", Diploma Thesis, University of Applied Sciences Kaiserslauten, Department of Computer Science and Micro-System Engineering, Zweibrücken, July 2003.

[FPL] Frame-Processing-Language, http://sourceforge.net/projects/fpl, last accessed in August 2012.

[FT] Frame Technology, http://www.servinghistory.com/topics/Frame_Technology_(software_engineering), last accessed in August 2012.

[GAL05] Gallardo, M.M., Martínez, J., Merino, P., Rodriguez, G.: "Integration of Reliability and Performance Analyses for Active Network Services", in: Electronic Notes in Theoretical Computer Science, vol. 133, pp. 217–236, 2005.

[GAMM95] Gamma E., Helm R., Johnson R., Vlissides J.: "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995.

[GER02] Gerber, A., Lawley, M., Raymond, K., Steel, J., Wood, A.: Transformation: "The Missing Link of MDA", In Lecture notes in computer science (2505), Springer-Verlag, 2002.

[GER03-a] Gerber, A., Raymond, K.: "MOF to EMF: There and Back Again", Cooperative Research Centre for Enterprise Distributed Systems (DSTC) Technical Report, University of Queensland, Australia, 2003.

[GER03-b] Gerber, A., Raymond, K.: "Mapping MOF to EMF", Cooperative Research Centre for Enterprise Distributed Systems (DSTC) Technical Report, University of Queensland, Australia, March 2003.

[GOM00] Gomaa, H.: "Designing Concurrent, Distributed, and Real-Time Applications with UML", Addison Wesley Longman Publishing Co., Inc., Boston, MA, 2000.

[GOM02] Gomaa, H., Shin, M. E.: "Multiple-View Meta-Modeling of Software Product Lines", the Eighth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2002), Maryland, December, p. 238, 2002.

[GOM04-a] Gomaa, H., Shin, M. E.: "A Multiple-View Meta-Modeling Approach for Variability Management in Software Product Lines", Proceeding of International Conference on Software Reuse, Madrid, Spain, Springer LNCS 3107, pp. 274-285, July 2004.

[GOM04-b] Gomaa, H.: "Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures", Addison-Wesley Longman Publishing Co., Inc., Redwood City, CA, 2004.

[GOM04-c] Gomaa, H., Webber, D.: "Modeling Adaptive and Evolvable Software Product Lines Using the Variation Point Model", In Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04), pp. 90268c - Track 9, 2004.

[GOM05] Gomaa, H.: "Designing Software Product Lines with UML: From Use Cases to Pattern-based software Architectures", Addison-Wesley Object Technology Series, 2005.

[GOM06-a] Gomaa, H.: "A Software Modeling Odyssey: Designing Evolutionary Architecture-centric Real-Time Systems and Product Lines", Keynote paper, Proc. 9th International Conference on Model-Driven Engineering, Languages, and Systems (MoDELS), Genova, Italy, pp. 1-15, October 2006.

[GOM06-b] Gomaa, H., Saleh, M.: "Feature Driven Dynamic Customization of Software Product Lines", Proc. Intl. Conf. on Software Reuse (ICSR), Torino, Italy, Springer LNCS 4039, pp. 58-72, June 2006.

[GOM07] Gomaa, H., Shin, M. E.: "Automated Software Product Line Engineering and Product Derivation", The 40th Annual Hawaii International Conference on System Sciences, Big Island, Hawaii, p. 285a, 2007.

[GRI06] Grieskamp, W.: "Multi-Paradigmatic Model-Based Testing", Technical Report #MSR-TR-2006-111, Microsoft Research, August 2006.

[GRIS00] Griss, M.: "Implementing Product-Line Features with Component Reuse", Proceedings of 6th International Conference on Software Reuse, Vienna, Austria, pp. 137-152, June 2000.

[GRO07-a] Groher, I., Papajewski, H., Voelter, M. : "Integrating Model-Driven Development and Software Product Line Engineering", Eclipse Summit, Ludwigsburg, Germany, 2007, available from http://citeseerx.ist.psu.edu/, last accessed in August 2012.

[GRO07-b] Groher, I., Voelter, M.: "Expressing Feature-Based Variability in Structural Models", *In Proceedings of the Workshop on Managing Variability for Software Product Lines,* SPLC, Kyoto, Japan, 2007, available from http://www.voelter.de, last accessed in August 2012.

[GRON05] Gronmo, R., Oldevik, J.: "An Empirical Study of the UML Model Transformation Tool (UMT)", In: INTEROP-ESA'05, Feb. 2005, available from http://folk.uio.no/roygr/, last accessed in August 2012.

[HAU04] Haugen, Ø., MOller-Pedersen, B., Oldevik, J., Solberg, A.: "An MDA-based framework for model-driven product derivation", In: M. H. Hamza, editor, Software Engineering and Applications, pp. 709-714, ACTA Press, Cambridge, 2004.

[HAU08] Haugen, Ø., Møller-Pedersen, B, B., Oldevik, J., Olsen, G., Svendsen, A.: "Adding standardized variability to domain specific languages", In: Proceedings of the 12th International Software Product Line Conference (SPLC08), IEEE (2008), pp. 139-148, 2008.

[HEI00-a] Hein, A., MacGregor, J., Schlick, M.: "Requirements and feature management for software product lines", in: Deutscher Software-Produktlinien Workshop (DSPL-1), Kaiserslautern, Germany, 2000.

[HEI00-b] Hein, A., Schlick, M., Vinga-Martins, R.: "Applying Feature Models in Industrial Settings", In Software Product Lines: Experience and Research Directions, Proceedings of the First Software Product Line Conference (SPLC1), P. Donohoe, Kluwer Academic Publishers, pp. 47-70, 2000.

[HEID07] Heidenreich., F., Wende, C.: "Bridging the gap between features and models", In the 2nd Workshop on Aspect-Oriented Product Line Engineering (AOPLE'07) co-located with the 6th International Conference on Generative Programming and Component Engineering (GPCE'07), Salzburg, Austria, October 2007, Online Proceedings URL: http://www.softeng.ox.ac.uk/aople/, last accessed in October 2012.

[HEID08-a] Heidenreich, F., Kopcsek, J., Wende, C.: "FeatureMapper: Mapping Features to Models", In: Companion Proceedings of the 30th International Conference on Software Engineering (ICSE08), New York, NY, USA, ACM, pp. 943-944, May 2008.

[HEID08-b] F. Heidenreich, I. Savga, and C. Wende, "On Controlled Visualisations in Software Product Line Engineering", In Proceedings of the 2nd International Workshop on Visualisation in Software Product Line Engineering (ViSPLE'08), collocated with the 12th International Software Product Line Conference (SPLC'08), pp. 335-341, September 2008.

[HEID09] F. Heidenreich, "Towards Systematic Ensuring Well-Formedness of Software Product Lines", In Proceedings of the 1st International workshop on Feature-Oriented Software Development (FOSD'09), pp. 69-74, Denver, Colorado, USA, 2009.

[HEID10] Heidenreich, F., Sanchez, P., Santos, J., Zschaler, S., Alferez, M., Araujo, J., Fuentes, L., Kulesza, U., Moreira, A., Rashid, A.: "Relating Feature Models to Other Models of a Software Product Line: A Comparative Study of FeatureMapper and VML*", Transactions on Aspect-Oriented Software Development (TAOSD), 6210, pp. 69-114, 2010.

[HELM90] 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.

[HOL92] 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.

[HTG] Houston Technology Group, http://www.theserverside.com/, last accessed in August 2012.

[IVK04] Ivkovic, I., Kontogiannis, K.: "Model Synchronization as a Problem of Maximizing Model Dependencies", In Proceedings of the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004), Vancouver, BC, Canada, pp. 222-223, October 2004.

[IVK06] Ivkovic, I., Kontogiannis, K.: "Towards Automatic Establishment of Model Dependencies Using Formal Concept Analysis", Accepted to the International Journal of Software Engineering and Knowledge Engineering (IJSEKE), (16)4: pp. 499-522, 2006.

[JAC97] Jacobson, I., Griss, M., Jonsson, P.: "Software Reuse-Architecture, Process and Organisation for Business Success", Addison-Wesley, Reading, Massachusetts, 1997.

[JARI04] Jaring, M., Bosch, J.: "A Taxonomy and Hierarchy of Variability Dependencies in Software Product Family Engineering", Proceeding of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04), IEEE, pp. 356-361, 2004.

[JAR01] Jarzabek, S., Zhang, H.: "Xml-based method and tool for handling variant requirements in domain models", In Procedings: 5th IEEE International Symposium on Requirements Engineering, RE 2001, pp. 166–173, 2001.

[JAR07] Jarzabek, S.: "Software Reuse beyond Components with XVCL (Tutorial)", Generative and Transformational Techniques in Software Engineering II: International Summer School, GTTSE 2007, Braga, Portugal, (LNCS 5235, Springer-Verlag, 2008, pp. 47-77), July 2007.

[JAR08] Jarzabek, S.: "Software Reuse beyond Components with XVCL", 2008, http://xvcl.comp.nus.edu.sg/overview_brochure.php, last accessed in August 2012.

[JAVA-a] aMOF2.0forJava, http://www2.informatik.hu-berlin.de/sam/meta-tools/aMOF2.0forJava/index.html, last accessed in August 2012.

[JAVA-b] JavaML, Java Machine Learning Library, http://www.badros.com/greg/JavaML/, last accessed in August 2012.

[JMI] Java Metadata Interface, http://java.sun.com/products/jmi/, last accessed in August 2012.

[JOST] Jostraca, http://www.jostraca.org/, last accessed in August 2012.

[JOU05] Jouault, F.: "Loosely Coupled Traceability for ATL", In Proceedings of the European Conference on Model Driven Architecture (ECMDA) Workshop on Traceability, Nuremberg, Germany, pp. 29-37, 2005.

[KAN90] 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.

[KEE99] Keepence B., Mannion M.: "Using Patterns to Model Variability in Product Families", IEEE Software, Vol. 16, No. 4, pp. 102-108, July/August 1999.

[KUR02] Kurtev, I., Bézivin, J., Aksit, M.: "Technological Spaces: An Initial Appraisal", In International Conference on Cooperative Information Systems (CoopIS), DOA'2002 Federated Conferences, Industrial Track , Irvine, USA. pp. 1-6, 30 Oct - 1 Nov 2002.

[LAU05] Lauenroth, K., Pohl, K.: "Principles of Variability", In: Pohl, K., Böckle, G., Linden, F.v.d. (eds.): Software Product Line Engineering, Foundations, Principles and Techniques, Springer, ISBN: 978-3-540-28901-2, pp. 57-88 2005.

[LAH07] Lahire, P., Morin, B., Vanwormhoudt, G., Gaignard, A., Barais, O., J'ez'equel, J.M.: "Introducing Variability into Aspect-Oriented Modeling Approaches", In Proceedings of ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS'07), LNCS, Nashville TN USA, Vanderbilt University, Springer-Verlag, pp. 498–513, 2007.

[LAW05] Lawley, M., Steel, J.: "Practical declarative model transformation with tefkat", In Proceedings of Satellite Events at the MoDELS 2005 Conference, Lecture Notes in Computer Science Vol. 3844, pp. 139-150, 2005.

[MAD02] Madhavan, J., Bernstein, P. A., Domingos, P., Halevy, A.Y.: "Representing and reasoning about mappings between domain models", In Eighteenth national conference on Artificial intelligence, American Association for Artificial Intelligence, pp. 80–86, 2002.

[MDA] Model Driven Architecture, Object Management Group, www.omg.org/mda/, last accessed in October 2012.

[McG05] McGregor, J.: "Ideas from SPLC", In Journal of Object Technology (JOT), vol. 4, no. 9, pp. 23-29, November-December 2005, http://www.jot.fm/issues/, last accessed in August 2012.

[McG07] McGregor, J.: "CM - Configuration Change Management", In Journal of Object Technology (JOT), vol. 6, no. 1, pp. 7-15, January 2007, http://www.jot.fm/issues/, last accessed in August 2012.

[MEY92] Meyer, B.: "Design by Contract", In *IEEE Computer*, vol. 25, no. 10, pp. 40-51, IEEE Press, New York, October 1992.

[MEY94] Meyer, B.: "Reusable Software: The Base object-oriented Component Libraries", ISBN-10: 0132454998, Prentice-Hall, 1994.

[MEY97] Meyer, B.: "Object-Oriented Software Construction", second edition, Prentice Hall, ISBN-10: 0136291554, 1997.

[MEZ07] Meszaros, G.: "xUnit Test Patterns: Refactoring Test Code", Addison-Wesley Professional, ISBN-10: 0131495054, 2007.

[MOF] Meta Object Facility (MOF) Specification, Technical Report, Object Management Group (OMG), April 2002, http://www.omg.org/mof/, last accessed in October 2012.

[MOO07] Moon, M., Chae, H. S., Nam, T., Yeom, K.: "A metamodeling approach to tracing variability between requirements and architecture in software product lines", In CIT'2007: Proc. of the 7th

IEEE International Conference on Computer and Information Technology, University of Aizu, Fukushima, Japan, pp. 927-933, 2007.

[MORI07] Morin, B., Barais, O., Jézéquel, J.M., Ramos, R.: "Towards a Generic Aspect-Oriented Modeling Framework", In Proceedings of the 3rd International Workshop on Models and Aspects (In conjunction of ECOOP'07), Berlin, Germany, 2007.

[MORI08] Morin, B., Barais, O., Jézéquel, J.M.: "Weaving Aspect Configurations for Managing System Variability", In 2nd International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'08), pp. 53-62, 2008.

[MOR08] Moros, B., Vicente-Chicote, C., Toval, A. : "Metamodeling Variability to Enable Requirements Reuse", 13th International Workshop on Exploring Modeling Methods for Systems Analysis and Design (EMMSAD'08), CEUR Workshop Proc. Vol. 337, pp. 140-154, 2008.

[MTF] Model Transformation Framework, IBM, http://www.alphaworks.ibm.com/tech/mtf, last accessed in August 2012.

[MUSS96] Musser, D., Saini, A.: "STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library", Addison-Wesley, ISBN-10: 0321702123, 1996.

[MYL02] Myllymäki, T.: "Variability Management in Software Product Lines", TTKK/OHJ Laboratory Technical Report 30, Institute of Software Systems, Tampere University of Technology, 2002.

[MYLO01] Mylopoulos, J., Chung, L., Liao, S., Wang, H., Yu, E.: "Exploring Alternatives during Requirements Analysis", IEEE Software, vol. 18, no.1, pp: 92- 96, Jan. 2001.

[oAW] openArchitectureWare (oAW) website, http://www.eclipse.org/gmt/oaw/, last accessed in August 2012.

[OCL] Object Constraint Language, Object Management Group, http://www.omg.org/technology/documents/modeling_spec_catalog.htm#OCL, last accessed in August 2012.

[OMG] Object Management Group, www.omg.org, last accessed in October 2012.

[OMG06] PIM and PSM for Software Radio Components, Object Management Group, http://sbc.omg.org/swradio_info.htm, last accessed in August 2012.

[ONO02] Ono, K., Koyanagi, T., Abe, M., Hori, M.: "Xslt stylesheet generation by example with wysiwyg editing," in SAINT '02: Proceedings of the 2002 Symposium on Applications and the Internet, Washington, DC, USA: IEEE Computer Society, pp. 150–161, 2002.

[OSEK01] OSEK/VDX System Generation, OIL: OSEK Implementation Language Version 2.3, September 10th, 2001, http://www.osek-vdx.org/, last accessed in August 2012.

[PAR76] Parnas, D.L.: "On the Design and Development of Program Families", IEEE Trans. Software Engineering, vol. 2, no, 1, pp. 1-9, March 1976.

[PEL01] Peltier, M., Bézevin, J., Guillaume, G.: "MTRANS: A general framework, based on XSLT for model transformations", In WTUML '01, Proceedings of the workshop on Transformations in UML, Genova, Italy, pp. 93-97, 2001.

[POH05] Pohl, K., Böckle, G., van der Linden, F.: "Software Product Line Engineering: Foundations, Principles, and Techniques", Springer, Berlin, Heidelberg, New York, ISBN: 3540243720, 2005.

[POH06] Pohl, K., Metzger, A.: "Variability Management in Software Product Line Engineering", Proceedings of the 29th ACM International Conference on Software Engineering (ICSE'06), Shanghai, China, pp. 186-187, May 20-28, 2006.

[POL02] Pollet, D., Vojtisek, D.: "OCL as a Core UML Transformation Language", In Workshop on Integration and Transformation of UML models (WITUML), in conjunction with European Conference on Object-oriented Programming (ECOOP 2002), Malaga, Spain, 2002.

[PURE] Pure: Variant Management Tool website, http://www.pure-systems.com/3.0.html, last accessed in August 2012.

[QUIC] Quick, http://jxquick.sourceforge.net/, last accessed in August 2012.

[QVT] Query/View/Transformations, Object Management Group, http://www.omg.org/spec/QVT/, last accessed in August 2012.

[RAB07] Rabiser, R., Dhungana, D., Grunbacher, P., Lehner, K., Federspiel, F.: "Involving non-technicians in product derivation and requirements engineering: a tool suite for product line engineering", in: 15th IEEE International Requirements Engineering Conference (RE'07), Delhi, India, pp. 367-368, October 2007.

[RAD11-a] Radonjic, V. D., Bashardoust-Tajali, S., Corriveau, J. P., Arnold, D.: "Implementing a Model for Design Pattern Selection and Evaluation Using ACL and the Validation Framework", IEEE/IFIP, Architecture-Based Testing and System Validation Workshop, WICSA2011, Boulder, Colorado, USA, 2011.

[RAD11-b] Radonjic, V. D., Bashardoust-Tajali, S., Corriveau, J. P., Arnold, D.: "Design Patterns –A Modeling Challenge", The 2011 International Conference on Software Engineering Research and Practice, SERP'11, Las Vegas, Nevada USA, 2011.

[RAS] Reusable Asset Specification (RAS), Object Management Group, http://www.omg.org/spec/RAS/, last accessed in August 2012.

[ROH05] Röhl, M., Uhrmacher, A.: "Flexible integration of XML into modeling and simulation systems", In Proceedings of 2005 Winter Simulation Conference (WSC), Orlando, Florida, USA, pp. 1813-1820, December 2005.

[RUP] Rational Unified Process (RUP), IBM Corporation, 2004, http://www.ibm.com/software/awdtools/rup/, last accessed in August 2012.

[SCH06] Schobbens, P.-Y., Heymans, P., Trigaux, J.-C., Bontemps, Y.: "Feature Diagrams: A Survey and A Formal Semantics", In: RE 2006, Proceedings of 14th IEEE International Requirements Engineering Conference, Minneapolis, IEEE Computer Society, Los Alamitos, pp. 139–148, 2006.

[SCHL00] Schlick, M., Hein, A.: "Knowledge engineering in software product lines", In Proceedings of the 14th European Conference on Artificial Intelligence (ECAI 2000), Workshop on Knowledge-Based Systems For Model-Based Engineering, Berlin, Germany, 2000.

[SHE02] Shehata, M.S., Eberlein, A., Hoover, H.J.: "Requirements Reuse and Feature Interaction Management", In Proceedings of the 15th International Conference on Software and Systems Engineering and their Applications (ICSSEA'02), Paris, France, 2002.

[SMA00] Smaragdakis, Y., Batory, D.: "Application generators", In: Webster, J. (ed.) Software Engineering volume of the Encyclopedia of Electrical and Electronics Engineering, John Wiley and Sons, Chichester, 2000.

[SMA02] Smaragdakis Y., Batory D.: "Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs", ACM Transactions on Software Engineering and Methodology, Vol. 11, No. 2, pp. 215-255, April 2002.

[SOUR] SourceCafe, http://www.sourcecafe.com/, last accessed in August 2012.

[STA06] Stahl, T., Voelter, M.: "Model-Driven Software Development", Wiley & Sons, 2006.

[STL95] Stepanov, A., STL Documentation, Silicon Graphics, Inc., http://www.sgi.com/Technology/STL, last accessed in October 2012.

[STO05] Stolz, V., Bodden, E.: "Temporal Assertions using AspectJ", In Fifth Workshop on Runtime Verfication (RV'05), Electronic Notes in Theoretical Computer Science (ENTCS), Elsevier Science Publishers, pp. 109-124, 2005.

[SVA00] Svahnberg, M., Gurp, J., Bosch, J.: "Research report: On the Notion of Variability in Software Product Lines", Blekinge Institute of Technology, Research Report 2000: 2, ISSN: 1103-1581, 2000.

[SVA02] Svahnberg, M., Gurp, J., Bosch, J.: "A Taxonomy of Variability Realization Techniques", Technical Paper 2002, ISSN: 1103-1581, Blekinge Institute of Technology, Sweden, (Published in Journal: Software-Practice & Experience, Vol. 35, No. 8, July 2005), pp. 705-754, 2002.

[TAV07] Tavakoli Kolagari, R., Reiser, M.O.: "Reusing requirements: the need for extended variability models", in: Intl. Symposium on Fundamentals of Software Engineering (FSEN 2007), Tehran, Iran, pp. 129-143, 2007.

[TAW12] Tawhid, R.: "Integrating Performance Analysis in Model Driven Software Product Line Engineering", Doctoral dissertation, School of Computer Science, Carleton University, May 2012.

[THI02-a] Thiel, S., Hein, A.: "Modeling and Using Product Line Variability in Automotive Systems", IEEE Software, p. 66ff, July/August 2002.

[THI02-b] Thiel, S., Hein, A.: "Systematic Integration of Variability into Product Line Architecture Design", In Proceedings of the 2nd International Software Product Lines Conference (SPLC2), Springer-Verlag, Berlin, pp. 130-153, 2002.

[TIA05] 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.

[UCM] Use Case Maps, http://jucmnav.softwareengineering.ca/twiki/bin/view/UCM/WebHome, last accessed in August 2012.

[UML] Unified Modeling Language Version 2.0, Object Management Group, http://www.omg.org/spec/UML/2.0/, last accessed in October 2012.

[UML2] Miles, R., Hamilton, K.: "Learning UML 2.0", O'Reilly, 2006.

[UMT] UML Model Transformation, http://umt-qvt.sourceforge.net/, last accessed in August 2012.

[UNIX] Unix Operating System, http://www.unix.org/, last accessed in August 2012.

[vanG01] van Gurp, J., Bosch, J., Svahnberg, M.: "On the Notion of Variability in Software Product Lines", In: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA 2001), pp. 45-54, 2001.

[VIN07] Vicente-Chicote, C., Moros, B., Toval, A.: "REMM-Studio: an Integrated Model-Driven Environment for Requirements Specification", Validation and Formatting, In Journal of Object Technology (JOT), Special Issue TOOLS EUROPE 2007, Vol. 6, No. 9, pp. 437-454, October 2007, http://www.jot.fm/issues/issue_2007_10/paper22/, last accessed in August 2012.

[VIS03] Visser E., Dolstra E., Florijn G.: "Timeline Variability: The variability of Binding Time of Variation Points", Proceedings of the Workshop on Software Variability Management, pp. 119-122, 2003.

[VOE07] Voelter, M., Groher, I.: "Handling Variability in Model Transformations and Generators", In Proceedings of the 7th European conference on object-oriented programming systems, languages, and applications, Conference (OOPSLA) Workshop on Domain-Specific Modeling (DSM'07), Montreal, Canada, (Published in Theory and Practice of Model Transformations, Springer, p. 198), 2007.

[WAN02] Wang, Y.-H., Lu, Y. C.: "An XML-based DEVS modeling tool to enhance simulation interoperability", In Proceedings of the 14th European Simulation Symposium (ESS'02), p. 406, 2002.

[WEB04] Webber, D. L., Gomaa, H.: "Modeling variability in software product lines with the variation point model, Science of Computer Programming", v.53 n.3, pp. 305-331, December 2004.

[XFRA] XFrames, http://www.w3.org/TR/xframes/, last accessed in October 2012.

[XMI] XML Metadata Interchange, Object Management Group, http://www.omg.org/spec/XMI/, last accessed in October 2012.

[XML-a] Extensible Markup Language, http://www.w3schools.com/xml/default.asp, last accessed in October 2012.

[XML-b] XML description, O'Reilly XML.com, http://www.xml.com/pub/pt/7, last accessed in October 2012.

[XML-c] XML-based code generators, http://www.xml.com/, last accessed in October 2012.

[XQU] Xquery requirements, http://www.w3.org/TR/xquery-30-requirements/, last accessed in August 2012.

[XSD] XML Schema Definition, http://www.w3.org/TR/xmlschema-0/, last accessed in August 2012.

[XSL-a] XSL Style-sheet, http://www.w3schools.com/xsl/default.asp, last accessed in October 2012.

[XSL-b] XSLT description, http://www.w3.org/TR/xslt20/, last accessed in October 2012.

[XSL-c] Mangano, S.: "Solutions and Examples for XML and XSLT Developers", O'Reilly Media, 2002.

[XVCL] http://xvcl.comp.nus.edu.sg/, last accessed in October 2012.

[ZIA03-a] Ziadi, T., Hélouët,L., Jézéquel,J. M.: "Towards a UML Profile for Software Product Lines", In Software Product-Family Engineering (SPFE 03), the 5th International Workshop, Published in Lecture Notes in Computer Science (LNCS), pp. 129–139, Springer, 2003.

[ZIA03-b] Ziadi, T., Jézéquel, J. M., Fondement, F.: "Product line derivation with uml", In Jilles van Gurp and Jan Bosh, editors, Proceedings Software Variability Management Workshop, University of Groningen Department of Mathematics and Computing Science, pp. 94–102, 2003.

[ZIA04] Ziadi, T., Hélouët, L., Jézéquel, J. M.: "Behaviors generation from product lines requirements", In Proceedings of the UML 2004 workshop on Software Architecture Description, Lisbon, Portugal, (Published in Software Product Lines: Research Issues in Engineering and Management, Springer, p. 477), September 2004.

[ZIA06] Ziadi, T., Jézéquel, J. M.: "Software Product Lines, chapter Product Line Engineering with the UML: Deriving Products", Springer, Online ISBN: 978-3-540-33252-7, pp. 557-586, 2006.

[ZIA07] Ziadi, T., Jézéquel, J. M.: "Plibs: an eclipse-based tool for software product line behavior engineering", In Proceedings of the 3rd Workshop on Managing Variability for Software Product Lines (SPLC 2007), Kyoto, Japan, 2007.

[ZHA04] 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.

[ZSCH09] Zschaler, S., S´anchez, P., Santos, J., Alf´erez, M., Rashid, A., Fuentes, L., Moreira, A., Ara´ujo, J., Kulesza, U.: "VML* – a family of languages for variability management in software product lines", In Gray, J., van den Brand, M., eds.: Proceedings of the 2nd International Conference on Software Language Engineering (SLE'09), Springer, pp. 82-102, 2009.

A.1 Grammar for Variability Relationships in the Relationships Table (RT)

```
RT-Statement → RT-depends-Relation|RT-requires-Relation
RT-depends-Relation →
Variability depends-relation Variability-list Cond-part-d
Action-part-d
|[VariationPoint←Feature-rel Variants]+
Cond-part-d → "Cond = -"
Action-part-d → "Action = variant"
RT-requires-Relation → Variability requires-relation Variability
Cond-part Action-part
| Relation-cond IMPLIES Cond
Cond-part → "Cond" Cond-number "=" Cond
Action-part → "Action = variant" {Cond-number}
Relation-cond → if (Cond-exp)
Cond-exp → Cond | cond-exp logical-op Cond-exp
Cond → VariationPoint rel-op Variants-n
Cond-number → INT
Variability-list → Variability {,Variability-list}
Variability → VariationPoint | Variants
depends-relation → "<depends>"
requires-relation → "<requires>"
Feature-rel → and-rel | or-rel | alternative
VariationPoint → VP-code Feature-type
Variants → Variants-code Feature-type
Variants-n → Variants-code Feature-type|Variants-name|null
VP-code → VP INT
Variants-code → VP INT-INT
Variants-name → String
logical-op → "or" | "and"
rel-op → "==" | "!="
and-type → " "
or-type → "R"
alternative → "X"
Feature-type → Mandatory | Optional
Mandatory → " "
Optional → "?"
INT → digits+
digits → 0-9
```

A.2 Grammar for Variability Contracts

```
Variability contracts → VP-statement+
VP-statement → "{"  VP-attributes [Variants-statement]+ "}"
VP-attributes → "Variation" VP-code "<"VP-name">" "["VP-option"]" "outof" VP-out
Variants-statement → Variants-attributes {Conditions}* Variants-def
Variants-attributes → "Case" Variants-name ":" "plug-in:" Variants-code
Conditions → Cond-name ":(" Cond-expr ")"
Cond-expr → Conds | Cond-expr logical-op Cond-expr
Conds → VP-code rel-op Variants-name
Variants-def → intraVariant-statement+
intraVariant-statement → interaVariant-attributes intraVariant-def
intraVariant-attributes → Plugin-code: Plugin-type
plugin-code → "Refine-" [a | r]
plugin-type → "Declaraion"| "Observability"| "Responsibility"| "Exports"|
 "Scenario"
intraVariant-def → Variants-heads "{" Variants-body+ "}"
Variants-heads → Variants-head, Variants-heads | Variants-head
Variants-head → head-type Plugin-name
Variants-body → Variants-part | Cond-part
Variants-part → "{" ACL-statement+ "}"
Cond-part → #IF (Cond-name) "{" Variants-part "}" #ENDIF
Cond-name → "Cond" Cond-number
Cond-number → INT
head-type → ACL-statement
Plugin-name → String
Variants-code → "VP" INT-INT
Variants-name → String
VP-code → "VP" INT
VP-name → String
VP-option → INT..INT
VP-out → INT
logical-op → "or" | "and"
rel-op → "==" | "!="
INT → digits+
digits → 0-9
```

A.3 Domain Contracts for the domain of Sequential Containers with selection templates: Linked-lists

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:import href="configlist.xsl"/>
<xsl:output method="text" indent="yes"/>


<xsl:template match="Contracts">

<xsl:text disable-output-escaping="yes">

Contract Iterator
  {
     Observability tItem getItem();
     Responsibility new()
     { Post(getItem() not= null); }
     Exports
     { Type tItem conforms Item; }
  }// End Contract

Contract Item
{
     Observability Integer Value();//returns value of item
     Responsibility new()
     { Post(Value() >= 0 ); }
     Responsibility finalize()
     { Pre(Value() >= 0); }
}// End Contract

abstract Contract ContainerBase &lt;Type T&gt;
{
     // Defining variables
     Scalar Integer v;
     Scalar Integer old_Size;
     Scalar Integer Size;

     // observabilities without bodies will be bound to correspondent IUT methods
     Observability INT size();
     Observability INT max_size();
     Observability Boolean IsEmpty();

     Observability Boolean IsFull()
     {
          </xsl:text>
          <xsl:apply-templates select="Variation1">
               <xsl:with-param name="param1" select="'Observability'" />
               <xsl:with-param name="param2" select="'IsFull'" />
          </xsl:apply-templates>
          <xsl:text disable-output-escaping="yes">
     }

     // new() will be bound to any type of constrcutors
     Responsibility new()
     {//allocate memory for container
          Size = 0;
          Post(IsEmpty()== true);
     }

     // finalize() will be bound to any type of destructors
     Responsibility finalize()
     { //free the allocated memory for container
          Pre(size() == 0);
          Pre(IsEmpty() == true);
     }

     Responsibility GenericInsertion()
     {//cannot insert into a full container
```

```
        Pre(IsFull() == false);
        old_Size = size();
        Execute();
        Size = Size + 1;
        Post(size() == old_Size + 1);
    }

    Responsibility GenericDeletion()
    {//cannot delete from empty container
        Pre(IsEmpty() == false);
        old_Size = size();
        Execute();
        Size = Size - 1;
        Post(size() == old_Size - 1);
    }

    Invariant SizeCheck
    {//Check if the Size is equal to the IUT container size
        Check(context.size >= 0);
        Check(context.Size == size());
    }

}// End Contract
</xsl:text>

<xsl:text disable-output-escaping="yes">
MainContract Container extends ContainerBase &lt;tItem&gt;
{
    Scalar tIterator tobeNext;
    Scalar tIterator tobePrev;
    Scalar tIterator new_pos;
    Scalar tIterator pos;

    // The size(), max_size(), and IsEmpty(), observabilities
    // will be copied to this contract
    Observability Integer Front();//returns value of first node
    Observability Integer Back();//returns value of last node
    Observability tIterator Begin();//returns pointer to first node
    Observability tIterator End();//returns pointer to last node
    Observability tItem itemAt(tIterator pos);//returns node
    Observability tIterator findNext(tIterator pos);//point to next

    //point to previous node
    //plugin_VP5();
        </xsl:text>
        <xsl:apply-templates select="Variation5">
            <xsl:with-param name="param1" select="'Declaration'" />
            <xsl:with-param name="param2" select="'find_prev'" />
        </xsl:apply-templates>
        <xsl:text disable-output-escaping="yes">

    //indexing an item in indexable container
    //plugin_VP8();
    </xsl:text>
    <xsl:apply-templates select="Variation8">
        <xsl:with-param name="param1" select="'Declaration'" />
        <xsl:with-param name="param2" select="'operatorIndex'" />
    </xsl:apply-templates>
    <xsl:text disable-output-escaping="yes">

    //tracing links for variability
    //plugin_VP33();
    </xsl:text>
    <xsl:apply-templates select="Variation33">
        <xsl:with-param name="param1" select="'Declaration'" />
        <xsl:with-param name="param2" select="'Dump-Forward-links'" />
    </xsl:apply-templates>
    <xsl:text disable-output-escaping="yes">

    //returning pointer to previous node
```

```
Observability tIterator find_prev(tIterator pos)
{
    //plugin_VP5();
    </xsl:text>
    <xsl:apply-templates select="Variation5">
        <xsl:with-param name="param1" select="'Observability'" />
        <xsl:with-param name="param2" select="'find_prev'" />
    </xsl:apply-templates>
    <xsl:text disable-output-escaping="yes">
}

//checking random access return in indexable container
Observability Boolean check_itemAt(Integer n)
{
    //plugin_VP7();
    </xsl:text>
    <xsl:apply-templates select="Variation7">
        <xsl:with-param name="param1" select="'Observability'" />
        <xsl:with-param name="param2" select="'check_itemAt'" />
    </xsl:apply-templates>
    <xsl:text disable-output-escaping="yes">
}

Observability Boolean in_range(Integer index)
{
    value = (index&gt;0 &amp;&amp; index&lt;=size());
}

Observability Boolean checkNext(tIterator pos, tIterator tobeNext)
{
    Boolean result=false;
    result = (findNext(pos) &amp;= tobeNext);
    value = result;
}

Observability Boolean checkPrev(tIterator pos, tIterator tobePrev)
{
    Boolean result=false;
    //plugin_VP5();
    result = (</xsl:text>
        <xsl:apply-templates select="Variation5"/>
        <xsl:text disable-output-escaping="yes">(pos) &amp;= tobePrev);
    value = result;
}

Parameters
{
  Scalar Boolean CheckMembes = true;
}

Structure
{
      choice(Parameters.CheckMembers) == true
      {
      //Check if a member of type tItem exist
      HasMemberOfType(tItem);
      }
  }

refine Responsibility new()
{
    Post(End() == Begin());
    //plugin_VP6();
    </xsl:text>
    <xsl:apply-templates select="Variation6">
        <xsl:with-param name="param1" select="'Responsibility'" />
        <xsl:with-param name="param2" select="'new'" />
    </xsl:apply-templates>
    <xsl:text disable-output-escaping="yes">
}
```

```
refine Responsibility finalize()
{
    fire(ContainerDone);
}

// Inserting item at the beginning of sequence
Responsibility Void Push_front(tItem item) extends GenericInsertion()
{
    tobeNext = Begin();
    Execute();
    Post(itemAt(Begin()) &amp;amp;= item);
    checkNext(Begin(), tobeNext);
    //plugin_VP4();
    </xsl:text>
    <xsl:apply-templates select="Variation4">
        <xsl:with-param name="param1" select="'Responsibility'" />
        <xsl:with-param name="param2" select="'Push_front'" />
    </xsl:apply-templates>
    <xsl:text disable-output-escaping="yes">
}

// Inserting item at the end of sequence
Responsibility Void Push_back(tItem item) extends GenericInsertion()
{
    tobeNext = End();
    tobePrev = find_prev(End());
    Execute();
    new_pos = find_prev(End());
    Post(itemAt(new_pos) == item);
    checkNext(new_pos, tobeNext);
    checkPrev(new_pos, tobePrev);
    //plugin_VP4();
    </xsl:text>
    <xsl:apply-templates select="Variation4">
        <xsl:with-param name="param1" select="'Responsibility'" />
        <xsl:with-param name="param2" select="'Push_back'" />
    </xsl:apply-templates>
    <xsl:text disable-output-escaping="yes">
}

// Inserting item before position
Responsibility tIterator Insert(tIterator pos, tItem item) extends GenericInsertion()
{   tobeNext = pos;
    tobePrev = find_prev(pos);
    Execute();
    Post(value not= null); //return-value from IUT insertion
    Post(itemAt(value) == item);
    checkNext(value, tobeNext);
    checkPrev(value, tobePrev);
    //plugin_VP4();
    </xsl:text>
    <xsl:apply-templates select="Variation4">
        <xsl:with-param name="param1" select="'Responsibility'" />
        <xsl:with-param name="param2" select="'Insert'" />
    </xsl:apply-templates>
    <xsl:text disable-output-escaping="yes">
}

// Inserting item after position
Responsibility tIterator InsertAfter(tIterator pos, tItem item) extends GenericInsertion()
{
    Pre(pos not= End());
    tobeNext = findNext(pos);
    tobePrev = pos;
    Execute();
    Post(value not= null); //return-value from insertAfter
    Post(itemAt(value) == item);
    checkNext(value, tobeNext);
    checkPrev(value, tobePrev);
```

```
        //plugin_VP4();
        </xsl:text>
        <xsl:apply-templates select="Variation4">
            <xsl:with-param name="param1" select="'Responsibility'" />
            <xsl:with-param name="param2" select="'InsertAfter'" />
        </xsl:apply-templates>
        <xsl:text disable-output-escaping="yes">
}

// Deleting item from the beginning of the sequence
Responsibility Void Pop_front() extends GenericDeletion()
{
        pos = Begin();
        new_pos = findNext(Begin());
        Execute();
        Post(Begin() == new_pos);
        //plugin_VP4();
        </xsl:text>
        <xsl:apply-templates select="Variation4">
            <xsl:with-param name="param1" select="'Responsibility'" />
            <xsl:with-param name="param2" select="'Pop_front'" />
        </xsl:apply-templates>
        <xsl:text disable-output-escaping="yes">
        //plugin_VP2();
        </xsl:text>
        <xsl:apply-templates select="Variation2">
            <xsl:with-param name="param1" select="'Responsibility'" />
            <xsl:with-param name="param2" select="'Pop_front'" />
        </xsl:apply-templates>
        <xsl:text disable-output-escaping="yes">
}

// Deleting item from the end of the sequence
Responsibility Void Pop_back() extends GenericDeletion()
{
        new_pos = find_prev(End());
        pos = new_pos;
        tobePrev = find_prev(new_pos);
        Execute();
        checkPrev(End(), tobePrev);
        //plugin_VP4();
        </xsl:text>
        <xsl:apply-templates select="Variation4">
            <xsl:with-param name="param1" select="'Responsibility'" />
            <xsl:with-param name="param2" select="'Pop_back'" />
        </xsl:apply-templates>
        <xsl:text disable-output-escaping="yes">
        //plugin_VP2();
        </xsl:text>
        <xsl:apply-templates select="Variation2">
            <xsl:with-param name="param1" select="'Responsibility'" />
            <xsl:with-param name="param2" select="'Pop_back'" />
        </xsl:apply-templates>
        <xsl:text disable-output-escaping="yes">
}

// Deleting item at position
Responsibility Void Erase(tIterator pos) extends GenericDeletion()
{
        Pre(pos not= End());
        tobeNext = findNext(pos);
        tobePrev = find_prev(pos);
        Execute();
        checkNext(tobePrev, tobeNext);
        checkPrev(tobeNext, tobePrev);
        //plugin_VP4();
        </xsl:text>
        <xsl:apply-templates select="Variation4">
            <xsl:with-param name="param1" select="'Responsibility'" />
            <xsl:with-param name="param2" select="'Erase'" />
```

```
    </xsl:apply-templates>
    <xsl:text disable-output-escaping="yes">
    //plugin_VP2();
    </xsl:text>
    <xsl:apply-templates select="Variation2">
        <xsl:with-param name="param1" select="'Responsibility'" />
        <xsl:with-param name="param2" select="'Erase'" />
    </xsl:apply-templates>
    <xsl:text disable-output-escaping="yes">
}

Responsibility Void resize()
{ //plugin_VP3();
</xsl:text>
<xsl:apply-templates select="Variation3">
    <xsl:with-param name="param1" select="'Responsibility'" />
    <xsl:with-param name="param2" select="'resize'" />
</xsl:apply-templates>
<xsl:text disable-output-escaping="yes">
}

Scenario ContainerLifetime
    {//Trigger when a new instance of the container ctor is created
     //Monitor any number of insertion and deletion in the container
     //Terminate when a container dtor is called
     Trigger(new()),
      (Push_front(dontcare) | Push_back(dontcare) |
    Insert(dontcare, dontcare) | InsertAfter(dontcare, dontcare))*,
      (pop_front() | Pop_back() | Erase(dontcare))*,
     observe(ContainerDone);
     Terminate(finalize());
    }

Metric Scalar Integer NumbeOfItems()
{
    context.Size;
}

Reports
{
    Report("A total of {0} items were created",
    NumberOfItems());
}

// exports section: to connect the container's Item type
Exports
{
    Type tItem conforms Item
    {
        not context;
        not derived context;
    }
    Type tIterator conforms Iterator;
    //plugin_VP0();
</xsl:text>
<xsl:apply-templates select="Variation0">
     <xsl:with-param name="param1" select="'Exports'" />
    <xsl:with-param name="param2" select="'key_type'" />
</xsl:apply-templates>
<xsl:text disable-output-escaping="yes">
    //plugin_VP30();
</xsl:text>
<xsl:apply-templates select="Variation30">
    <xsl:with-param name="param1" select="'Exports'" />
    <xsl:with-param name="param2" select="'length-counter'" />
</xsl:apply-templates>
<xsl:text disable-output-escaping="yes">
    //plugin_VP31();
</xsl:text>
<xsl:apply-templates select="Variation31">
```

```
            <xsl:with-param name="param1" select="'Exports'" />
            <xsl:with-param name="param2" select="'LC-Type'" />
        </xsl:apply-templates>
        <xsl:text disable-output-escaping="yes">
        }
}// End Contract

</xsl:text>
<xsl:text> </xsl:text>

</xsl:template>


<xsl:template match="Variation0">
  <xsl:param name="param1"/>
  <xsl:param name="param2"/>
  <xsl:for-each select="Case">
      <xsl:choose>
      <xsl:when test="@name=$VP0">
       <xsl:for-each select="intraVariant">
         <xsl:choose>
            <xsl:when test="@type=$param1 and @name=$param2">
                <xsl:value-of select="@variant"/>
            </xsl:when>
            <xsl:otherwise>
           <xsl:text></xsl:text>
            </xsl:otherwise>
        </xsl:choose>
       </xsl:for-each>
      </xsl:when>
      <xsl:otherwise>
             <xsl:text></xsl:text>
      </xsl:otherwise>
      </xsl:choose>
  </xsl:for-each>
</xsl:template>


 <xsl:template match="Variation1">
  <xsl:param name="param1"/>
  <xsl:param name="param2"/>
  <xsl:for-each select="Case">
      <xsl:choose>
      <xsl:when test="@name=$VP1">
       <xsl:for-each select="intraVariant">
         <xsl:choose>
            <xsl:when test="@type=$param1 and @name=$param2">
                <xsl:value-of select="@variant"/>
            </xsl:when>
            <xsl:otherwise>
           <xsl:text></xsl:text>
            </xsl:otherwise>
        </xsl:choose>
       </xsl:for-each>
      </xsl:when>
      <xsl:otherwise>
             <xsl:text></xsl:text>
      </xsl:otherwise>
      </xsl:choose>
  </xsl:for-each>
</xsl:template>


<xsl:template match="Variation2">
  <xsl:param name="param1"/>
  <xsl:param name="param2"/>
  <xsl:for-each select="Case">
      <xsl:choose>
      <xsl:when test="@name=$VP2">
       <xsl:for-each select="intraVariant">
```

```
          <xsl:choose>
             <xsl:when test="@type=$param1 and @name=$param2">
                 <xsl:value-of select="@variant"/>
             </xsl:when>
             <xsl:otherwise>
            <xsl:text></xsl:text>
             </xsl:otherwise>
          </xsl:choose>
        </xsl:for-each>
      </xsl:when>
      <xsl:otherwise>
             <xsl:text></xsl:text>
      </xsl:otherwise>
      </xsl:choose>
  </xsl:for-each>
</xsl:template>


<xsl:template match="Variation3">
   <xsl:param name="param1"/>
   <xsl:param name="param2"/>
   <xsl:for-each select="Case">
      <xsl:choose>
      <xsl:when test="@name=$VP3  and  $VP3='True' and $VP1='Fixed'">
       <xsl:for-each select="intraVariant">
        <xsl:choose>
             <xsl:when test="@type=$param1 and @name=$param2">
                 <xsl:value-of select="@variant1"/>
             </xsl:when>
             <xsl:otherwise>
             <xsl:text></xsl:text>
             </xsl:otherwise>
          </xsl:choose>
        </xsl:for-each>
      </xsl:when>
      <xsl:when test="@name=$VP3  and  $VP3='False'">
       <xsl:for-each select="intraVariant">
        <xsl:choose>
             <xsl:when test="@type=$param1 and @name=$param2">
                 <xsl:value-of select="@variant"/>
             </xsl:when>
             <xsl:otherwise>
             <xsl:text></xsl:text>
             </xsl:otherwise>
          </xsl:choose>
        </xsl:for-each>
      </xsl:when>
      <xsl:otherwise>
             <xsl:text></xsl:text>
      </xsl:otherwise>
      </xsl:choose>
  </xsl:for-each>
</xsl:template>


<xsl:template match="Variation4">
   <xsl:param name="param1"/>
   <xsl:param name="param2"/>
   <xsl:for-each select="Case">
      <xsl:choose>
      <xsl:when test="@name=$VP4  and  $VP4='Circular' and $VP5='Two-way'">
       <xsl:for-each select="intraVariant">
        <xsl:choose>
             <xsl:when test="@type=$param1 and @name=$param2">
                 <xsl:value-of select="@variant1"/>
             </xsl:when>
             <xsl:otherwise>
             <xsl:text></xsl:text>
             </xsl:otherwise>
          </xsl:choose>
```

```
      </xsl:for-each>
    </xsl:when>
    <xsl:when test="@name=$VP4  and  $VP4='Circular'  and  $VP5='One-way'">
      <xsl:for-each select="intraVariant">
       <xsl:choose>
          <xsl:when test="@type=$param1 and @name=$param2">
              <xsl:value-of select="@variant2"/>
          </xsl:when>
          <xsl:otherwise>
          <xsl:text></xsl:text>
          </xsl:otherwise>
       </xsl:choose>
      </xsl:for-each>
    </xsl:when>
    <xsl:when test="@name=$VP4  and  $VP4='Non-Circular'  and  $VP5='Two-way'">
      <xsl:for-each select="intraVariant">
       <xsl:choose>
          <xsl:when test="@type=$param1 and @name=$param2">
              <xsl:value-of select="@variant1"/>
          </xsl:when>
          <xsl:otherwise>
          <xsl:text></xsl:text>
          </xsl:otherwise>
       </xsl:choose>
      </xsl:for-each>
    </xsl:when>
    <xsl:when test="@name=$VP4  and  $VP4='Non-Circular'  and  $VP5='One-way'">
      <xsl:for-each select="intraVariant">
       <xsl:choose>
          <xsl:when test="@type=$param1 and @name=$param2">
              <xsl:value-of select="@variant2"/>
          </xsl:when>
          <xsl:otherwise>
          <xsl:text></xsl:text>
          </xsl:otherwise>
       </xsl:choose>
      </xsl:for-each>
    </xsl:when>
    <xsl:otherwise>
            <xsl:text></xsl:text>
    </xsl:otherwise>
    </xsl:choose>
  </xsl:for-each>
</xsl:template>


<xsl:template match="Variation5">
   <xsl:param name="param1"/>
   <xsl:param name="param2"/>
   <xsl:for-each select="Case">
      <xsl:choose>
        <xsl:when test="@name=$VP5">
    <xsl:for-each select="intraVariant">
       <xsl:choose>
          <xsl:when test="@type=$param1 and @name=$param2">
              <xsl:value-of select="@variant"/>
          </xsl:when>
          <xsl:otherwise>
          <xsl:text></xsl:text>
          </xsl:otherwise>
       </xsl:choose>
    </xsl:for-each>
      </xsl:when>
      <xsl:otherwise>
       <xsl:text></xsl:text>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:for-each>
</xsl:template>
```

```
<xsl:template match="Variation6">
   <xsl:param name="param1"/>
   <xsl:param name="param2"/>
   <xsl:for-each select="Case">
      <xsl:choose>
        <xsl:when test="@name=$VP6">
      <xsl:for-each select="intraVariant">
         <xsl:choose>
            <xsl:when test="@type=$param1 and @name=$param2">
                <xsl:value-of select="@variant"/>
            </xsl:when>
            <xsl:otherwise>
            <xsl:text></xsl:text>
            </xsl:otherwise>
         </xsl:choose>
      </xsl:for-each>
        </xsl:when>
        <xsl:otherwise>
         <xsl:text></xsl:text>
        </xsl:otherwise>
      </xsl:choose>
  </xsl:for-each>
</xsl:template>

<xsl:template match="Variation7">
   <xsl:param name="param1"/>
   <xsl:param name="param2"/>
   <xsl:for-each select="Case">
      <xsl:choose>
      <xsl:when test=
      "@name=$VP7  and  $VP7='True' and $VP8='Indexable' and $VP1='Fixed'">
       <xsl:for-each select="intraVariant">
         <xsl:choose>
            <xsl:when test="@type=$param1 and @name=$param2">
                <xsl:value-of select="@variant1"/>
            </xsl:when>
            <xsl:otherwise>
            <xsl:text></xsl:text>
            </xsl:otherwise>
         </xsl:choose>
        </xsl:for-each>
      </xsl:when>
      <xsl:when test="@name=$VP7  and  $VP7='False'">
        <xsl:for-each select="intraVariant">
         <xsl:choose>
            <xsl:when test="@type=$param1 and @name=$param2">
                <xsl:value-of select="@variant"/>
            </xsl:when>
            <xsl:otherwise>
           <xsl:text></xsl:text>
            </xsl:otherwise>
         </xsl:choose>
        </xsl:for-each>
          </xsl:when>
      <xsl:otherwise>
              <xsl:text></xsl:text>
      </xsl:otherwise>
      </xsl:choose>
  </xsl:for-each>
</xsl:template>

<xsl:template match="Variation8">
   <xsl:param name="param1"/>
   <xsl:param name="param2"/>
   <xsl:for-each select="Case">
      <xsl:choose>
      <xsl:when test="@name=$VP8  and  $VP8='Indexable' and  $VP1='Fixed'">
        <xsl:for-each select="intraVariant">
         <xsl:choose>
```

```
            <xsl:when test="@type=$param1 and @name=$param2">
                <xsl:value-of select="@variant1"/>
            </xsl:when>
            <xsl:otherwise>
            <xsl:text></xsl:text>
            </xsl:otherwise>
          </xsl:choose>
        </xsl:for-each>
      </xsl:when>
      <xsl:when test="@name=$VP8 and $VP8='Non-Indexable'">
        <xsl:for-each select="intraVariant">
          <xsl:choose>
            <xsl:when test="@type=$param1 and @name=$param2">
                <xsl:value-of select="@variant"/>
            </xsl:when>
            <xsl:otherwise>
           <xsl:text></xsl:text>
            </xsl:otherwise>
          </xsl:choose>
        </xsl:for-each>
          </xsl:when>
      <xsl:otherwise>
              <xsl:text></xsl:text>
      </xsl:otherwise>
      </xsl:choose>
  </xsl:for-each>
</xsl:template>

<xsl:template match="Variation30">
   <xsl:param name="param1"/>
   <xsl:param name="param2"/>
   <xsl:for-each select="Case">
      <xsl:choose>
      <xsl:when test="@name=$VP30 and ($VP1='Variable' or $VP1='Fixed')">
        <xsl:for-each select="intraVariant">
          <xsl:choose>
            <xsl:when test="@type=$param1 and @name=$param2">
                <xsl:value-of select="@variant1"/>
            </xsl:when>
            <xsl:otherwise>
           <xsl:text></xsl:text>
            </xsl:otherwise>
          </xsl:choose>
        </xsl:for-each>
      </xsl:when>
      <xsl:otherwise>
              <xsl:text></xsl:text>
      </xsl:otherwise>
      </xsl:choose>
  </xsl:for-each>
</xsl:template>

<xsl:template match="Variation31">
   <xsl:param name="param1"/>
   <xsl:param name="param2"/>
   <xsl:for-each select="Case">
      <xsl:choose>
      <xsl:when test="@name=$VP31 and $VP30='LC-Type' and ($VP1='Variable' or $VP1='Fixed')">
        <xsl:for-each select="intraVariant">
          <xsl:choose>
            <xsl:when test="@type=$param1 and @name=$param2">
                <xsl:value-of select="@variant1"/>
            </xsl:when>
            <xsl:otherwise>
           <xsl:text></xsl:text>
            </xsl:otherwise>
          </xsl:choose>
        </xsl:for-each>
      </xsl:when>
      <xsl:otherwise>
```

```
            <xsl:text></xsl:text>
        </xsl:otherwise>
        </xsl:choose>
    </xsl:for-each>
</xsl:template>

<xsl:template match="Variation33">
    <xsl:param name="param1"/>
    <xsl:param name="param2"/>
    <xsl:for-each select="Case">
        <xsl:choose>
        <xsl:when test="@name=$VP33">
          <xsl:for-each select="intraVariant">
            <xsl:choose>
                <xsl:when test="@type=$param1 and @name=$param2">
                    <xsl:value-of select="@variant"/>
                </xsl:when>
                <xsl:otherwise>
               <xsl:text></xsl:text>
                </xsl:otherwise>
            </xsl:choose>
          </xsl:for-each>
        </xsl:when>
        <xsl:otherwise>
                <xsl:text></xsl:text>
        </xsl:otherwise>
        </xsl:choose>
    </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

A.4 The derived MC for a single-linked-list container

```
Contract Iterator
  {
    Observability tItem getItem();
    Responsibility new()
    { Post(getItem() not= null); }
    Exports
    { Type tItem conforms Item; }
  }// End Contract

Contract Item
{
    Observability Integer Value();//returns value of item
    Responsibility new()
    { Post(Value() >= 0 ); }
    Responsibility finalize()
    { Pre(Value() >= 0); }
}// End Contract

abstract Contract ContainerBase <Type T>
{
    // Defining variables
    Scalar Integer v;
    Scalar Integer old_Size;
    Scalar Integer Size;

    // observabilities without bodies will be bound to correspondent IUT methods
    Observability INT size();
    Observability INT max_size();
    Observability Boolean isEmpty();

    Observability Boolean IsFull()
    {
        value = false;
    }

    // new() will be bound to any type of constrcutors
    Responsibility new()
    {//allocate memory for container
        Size = 0;
        Post(IsEmpty()== true);
    }

    // finalize() will be bound to any type of destructors
    Responsibility finalize()
    { //free the allocated memory for container
        Pre(size() == 0);
        Pre(IsEmpty() == true);
    }

    Responsibility GenericInsertion()
    {//cannot insert into a full container
        Pre(IsFull() == false);
        old_Size = size();
        Execute();
        Size = Size + 1;
        Post(size() == old_Size + 1);
    }

    Responsibility GenericDeletion()
    {//cannot delete from empty container
        Pre(IsEmpty() == false);
        old_Size = size();
        Execute();
        Size = Size - 1;
        Post(size() == old_Size - 1);
```

```
    }

    Invariant SizeCheck
    {//Check if the Size is equal to the IUT container size
        Check(context.size >= 0);
        Check(context.Size == size());
    }

}// End Contract

MainContract Container extends ContainerBase <tItem>
{
    Scalar tIterator tobeNext;
    Scalar tIterator tobePrev;
    Scalar tIterator new_pos;
    Scalar tIterator pos;

    // The size(), max_size(), and IsEmpty(), observabilities
    // will be copied to this contract
    Observability Integer Front();//returns value of first node
    Observability Integer Back();//returns value of last node
    Observability tIterator Begin();//returns pointer to first node
    Observability tIterator End();//returns pointer to last node
    Observability tItem itemAt(tIterator pos);//returns node
    Observability tIterator findNext(tIterator pos);//point to next

    //point to previous node
    //plugin_VP5();
        Observability tIterator Previous(tIterator pos);

    //indexing an item in indexable container
    //plugin_VP8();


    //tracing links for variability
    //plugin_VP33();


    //returning pointer to previous node
    Observability tIterator find_prev(tIterator pos)
    {
        //plugin_VP5();
        value = Previous(pos);
    }

    //checking random access return in indexable container
    Observability Boolean check_itemAt(Integer n)
    {
        //plugin_VP7();

    }

    Observability Boolean in_range(Integer index)
    {
        value = (index>0 && index<=size());
    }

    Observability Boolean checkNext(tIterator pos, tIterator tobeNext)
    {
        Boolean result=false;
        result = (findNext(pos) &= tobeNext);
        value = result;
    }

    Observability Boolean checkPrev(tIterator pos, tIterator tobePrev)
    {
        Boolean result=false;
        //plugin_VP5();
        result = ((pos) &= tobePrev);
        value = result;
```

```
}

Parameters
{
   Scalar Boolean CheckMembes = true;
}

Structure
{
        choice(Parameters.CheckMembers) == true
        {
        //Check if a member of type tItem exist
        HasMemberOfType(tItem);
        }
   }

refine Responsibility new()
{
     Post(End() == Begin());
     //plugin_VP6();
     Post(End() not= null);
    Post(itemAt(End()) == null);
    Post(size() == 1);
}

refine Responsibility finalize()
{
     fire(ContainerDone);
}

// Inserting item at the beginning of sequence
Responsibility Void Push_front(tItem item) extends GenericInsertion()
{
     tobeNext = Begin();
     Execute();
     Post(itemAt(Begin()) &= item);
     checkNext(Begin(), tobeNext);
     //plugin_VP4();
     tobeNext = null;
     Post(checkNext(End(), tobeNext) == true);
}

// Inserting item at the end of sequence
Responsibility Void Push_back(tItem item) extends GenericInsertion()
{
     tobeNext = End();
     tobePrev = find_prev(End());
     Execute();
     new_pos = find_prev(End());
     Post(itemAt(new_pos) == item);
     checkNext(new_pos, tobeNext);
     checkPrev(new_pos, tobePrev);
     //plugin_VP4();
     tobeNext = null;
     Post(checkNext(End(), tobeNext) == true);
}

// Inserting item before position
Responsibility tIterator Insert(tIterator pos, tItem item) extends GenericInsertion()
{    tobeNext = pos;
     tobePrev = find_prev(pos);
     Execute();
     Post(value not= null); //return-value from IUT insertion
     Post(itemAt(value) == item);
     checkNext(value, tobeNext);
     checkPrev(value, tobePrev);
     //plugin_VP4();
     tobeNext = null;
     Post(checkNext(End(), tobeNext) == true);
}
```

```
// Inserting item after position
Responsibility tIterator InsertAfter(tIterator pos, tItem item) extends GenericInsertion()
{
      Pre(pos not= End());
      tobeNext = findNext(pos);
      tobePrev = pos;
      Execute();
      Post(value not= null); //return-value from insertAfter
      Post(itemAt(value) == item);
      checkNext(value, tobeNext);
      checkPrev(value, tobePrev);
      //plugin_VP4();
      tobeNext = null;
      Post(checkNext(End(), tobeNext) == true);
}

// Deleting item from the beginning of the sequence
Responsibility Void Pop_front() extends GenericDeletion()
{
      pos = Begin();
      new_pos = findNext(Begin());
      Execute();
      Post(Begin() == new_pos);
      //plugin_VP4();
      tobeNext = null;
      Post(checkNext(End(), tobeNext) == true);
      //plugin_VP2();

}

// Deleting item from the end of the sequence
Responsibility Void Pop_back() extends GenericDeletion()
{
      new_pos = find_prev(End());
      pos = new_pos;
      tobePrev = find_prev(new_pos);
      Execute();
      checkPrev(End(), tobePrev);
      //plugin_VP4();
      tobeNext = null;
      Post(checkNext(End(), tobeNext) == true);
      //plugin_VP2();

}

// Deleting item at position
Responsibility Void Erase(tIterator pos) extends GenericDeletion()
{
      Pre(pos not= End());
      tobeNext = findNext(pos);
      tobePrev = find_prev(pos);
      Execute();
      checkNext(tobePrev, tobeNext);
      checkPrev(tobeNext, tobePrev);
      //plugin_VP4();
      tobeNext = null;
      Post(checkNext(End(), tobeNext) == true);
      //plugin_VP2();

}

Responsibility Void resize()
{ //plugin_VP3();

}

Scenario ContainerLifetime
      {//Trigger when a new instance of the container ctor is created
       //Monitor any number of insertion and deletion in the container
```

```
            //Terminate when a container dtor is called
            Trigger(new()),
             (Push_front(dontcare) | Push_back(dontcare) |
            Insert(dontcare, dontcare) | InsertAfter(dontcare, dontcare))*,
             (pop_front() | Pop_back() | Erase(dontcare))*,
            observe(ContainerDone);
            Terminate(finalize());
           }

    Metric Scalar Integer NumbeOfItems()
    {
        context.Size;
    }

    Reports
    {
        Report("A total of {0} items were created",
        NumberOfItems());
    }

    // exports section: to connect the container's Item type
    Exports
    {
        Type tItem conforms Item
        {
            not context;
            not derived context;
        }
        Type tIterator conforms Iterator;
        //plugin_VP0();

        //plugin_VP30();
            Type INT conforms Integer;
        //plugin_VP31();

    }
}// End Contract
```

```
Contract Iterator
  {
     Observability tItem getItem();
     Responsibility new()
     { Post(getItem() not= null); }
     Exports
     { Type tItem conforms Item; }
  }// End Contract

Contract Item
{
     Observability Integer Value();//returns value of item
     Responsibility new()
     { Post(Value() >= 0 ); }
     Responsibility finalize()
     { Pre(Value() >= 0); }
}// End Contract

abstract Contract ContainerBase <Type T>
{
     // Defining variables
     Scalar Integer v;
     Scalar Integer old_Size;
     Scalar Integer Size;

     // observabilities without bodies will be bound to correspondent IUT methods
     Observability INT size();
     Observability INT max_size();
     Observability Boolean IsEmpty();

     Observability Boolean IsFull()
     {
          value = false;
     }

     // new() will be bound to any type of constrcutors
     Responsibility new()
     {//allocate memory for container
          Size = 0;
          Post(IsEmpty()== true);
     }

     // finalize() will be bound to any type of destructors
     Responsibility finalize()
     { //free the allocated memory for container
          Pre(size() == 0);
          Pre(IsEmpty() == true);
     }

     Responsibility GenericInsertion()
     {//cannot insert into a full container
          Pre(IsFull() == false);
          old_Size = size();
          Execute();
          Size = Size + 1;
          Post(size() == old_Size + 1);
     }

     Responsibility GenericDeletion()
     {//cannot delete from empty container
          Pre(IsEmpty() == false);
          old_Size = size();
          Execute();
          Size = Size - 1;
          Post(size() == old_Size - 1);
```

```
    }

    Invariant SizeCheck
    {//Check if the Size is equal to the IUT container size
        Check(context.size >= 0);
        Check(context.Size == size());
    }

}// End Contract

MainContract Container extends ContainerBase <tItem>
{
    Scalar tIterator tobeNext;
    Scalar tIterator tobePrev;
    Scalar tIterator new_pos;
    Scalar tIterator pos;

    // The size(), max_size(), and IsEmpty(), observabilities
    // will be copied to this contract
    Observability Integer Front();//returns value of first node
    Observability Integer Back();//returns value of last node
    Observability tIterator Begin();//returns pointer to first node
    Observability tIterator End();//returns pointer to last node
    Observability tItem itemAt(tIterator pos);//returns node
    Observability tIterator findNext(tIterator pos);//point to next

    //point to previous node
    //plugin_VP5();
        Observability tIterator findPrev(tIterator pos);

    //indexing an item in indexable container
    //plugin_VP8();


    //tracing links for variability
    //plugin_VP33();


    //returning pointer to previous node
    Observability tIterator find_prev(tIterator pos)
    {
        //plugin_VP5();
        value = findPrev(pos);
    }

    //checking random access return in indexable container
    Observability Boolean check_itemAt(Integer n)
    {
        //plugin_VP7();

    }

    Observability Boolean in_range(Integer index)
    {
        value = (index>0 && index<=size());
    }

    Observability Boolean checkNext(tIterator pos, tIterator tobeNext)
    {
        Boolean result=false;
        result = (findNext(pos) &= tobeNext);
        value = result;
    }

    Observability Boolean checkPrev(tIterator pos, tIterator tobePrev)
    {
        Boolean result=false;
        //plugin_VP5();
        result = ((pos) &= tobePrev);
        value = result;
```

```
}

Parameters
{
   Scalar Boolean CheckMembes = true;
}

Structure
{
        choice(Parameters.CheckMembers) == true
        {
        //Check if a member of type tItem exist
        HasMemberOfType(tItem);
        }
   }

refine Responsibility new()
{
      Post(End() == Begin());
      //plugin_VP6();
      Post(End() not= null);
     Post(itemAt(End()) == null);
     Post(size() == 1);
}

refine Responsibility finalize()
{
      fire(ContainerDone);
}

// Inserting item at the beginning of sequence
Responsibility Void Push_front(tItem item) extends GenericInsertion()
{
      tobeNext = Begin();
      Execute();
      Post(itemAt(Begin()) &= item);
      checkNext(Begin(), tobeNext);
      //plugin_VP4();
      tobeNext = Begin();
      Post(checkNext(End(), tobeNext) == true);
       tobePrev = End();
       Post(checkPrev(Begin(), tobePrev) == true);
}

// Inserting item at the end of sequence
Responsibility Void Push_back(tItem item) extends GenericInsertion()
{
      tobeNext = End();
      tobePrev = find_prev(End());
      Execute();
      new_pos = find_prev(End());
      Post(itemAt(new_pos) == item);
      checkNext(new_pos, tobeNext);
      checkPrev(new_pos, tobePrev);
      //plugin_VP4();
      tobeNext = Begin();
      Post(checkNext(End(), tobeNext) == true);
       tobePrev = End();
       Post(checkPrev(Begin(), tobePrev) == true);
}

// Inserting item before position
Responsibility tIterator Insert(tIterator pos, tItem item) extends GenericInsertion()
{     tobeNext = pos;
      tobePrev = find_prev(pos);
      Execute();
      Post(value not= null); //return-value from IUT insertion
      Post(itemAt(value) == item);
      checkNext(value, tobeNext);
      checkPrev(value, tobePrev);
```

```
        //plugin_VP4();
        tobeNext = Begin();
        Post(checkNext(End(), tobeNext) == true);
         tobePrev = End();
         Post(checkPrev(Begin(), tobePrev) == true);
}

// Inserting item after position
Responsibility tIterator InsertAfter(tIterator pos, tItem item) extends GenericInsertion()
{
        Pre(pos not= End());
        tobeNext = findNext(pos);
        tobePrev = pos;
        Execute();
        Post(value not= null); //return-value from insertAfter
        Post(itemAt(value) == item);
        checkNext(value, tobeNext);
        checkPrev(value, tobePrev);
        //plugin_VP4();
        tobeNext = Begin();
        Post(checkNext(End(), tobeNext) == true);
         tobePrev = End();
         Post(checkPrev(Begin(), tobePrev) == true);
}

// Deleting item from the beginning of the sequence
Responsibility Void Pop_front() extends GenericDeletion()
{
        pos = Begin();
        new_pos = findNext(Begin());
        Execute();
        Post(Begin() == new_pos);
        //plugin_VP4();
        tobeNext = Begin();
        Post(checkNext(End(), tobeNext) == true);
         tobePrev = End();
         Post(checkPrev(Begin(), tobePrev) == true);
        //plugin_VP2();

}

// Deleting item from the end of the sequence
Responsibility Void Pop_back() extends GenericDeletion()
{
        new_pos = find_prev(End());
        pos = new_pos;
        tobePrev = find_prev(new_pos);
        Execute();
        checkPrev(End(), tobePrev);
        //plugin_VP4();
        tobeNext = Begin();
        Post(checkNext(End(), tobeNext) == true);
         tobePrev = End();
         Post(checkPrev(Begin(), tobePrev) == true);
        //plugin_VP2();

}

// Deleting item at position
Responsibility Void Erase(tIterator pos) extends GenericDeletion()
{
        Pre(pos not= End());
        tobeNext = findNext(pos);
        tobePrev = find_prev(pos);
        Execute();
        checkNext(tobePrev, tobeNext);
        checkPrev(tobeNext, tobePrev);
        //plugin_VP4();
        tobeNext = Begin();
        Post(checkNext(End(), tobeNext) == true);
```

```
            tobePrev = End();
            Post(checkPrev(Begin(), tobePrev) == true);
            //plugin_VP2();

        }

        Responsibility Void resize()
        { //plugin_VP3();

        }

        Scenario ContainerLifetime
            {//Trigger when a new instance of the container ctor is created
             //Monitor any number of insertion and deletion in the container
             //Terminate when a container dtor is called
             Trigger(new()),
              (Push_front(dontcare) | Push_back(dontcare) |
            Insert(dontcare, dontcare) | InsertAfter(dontcare, dontcare))*,
              (pop_front() | Pop_back() | Erase(dontcare))*,
             observe(ContainerDone);
             Terminate(finalize());
            }

        Metric Scalar Integer NumbeOfItems()
        {
            context.Size;
        }

        Reports
        {
            Report("A total of {0} items were created",
            NumberOfItems());
        }

        // exports section: to connect the container's Item type
        Exports
        {
            Type tItem conforms Item
            {
                not context;
                not derived context;
            }
            Type tIterator conforms Iterator;
            //plugin_VP0();

            //plugin_VP30();
                Type INT conforms Integer;
            //plugin_VP31();
        }
}// End Contract
```