Modeling Use-Case Sequential Dependencies using ACL

by

Inam Ul Haq

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

Ottawa-Carleton Institute for Computer Science
School of Computer Science

Carleton University
Ottawa, Ontario

Abstract

UML's support for modeling inter-use-case semantics is very limited. Briand and Labiche (2002) propose the TOTEM methodology to address the generation of tests for such dependencies. However, their generation algorithm, which heavily relies the notion of interleaving, presents one major problem: even for small examples, the number of sequences of use cases to test is unmanageable. The question we ask is: is it feasible to capture use cases and their sequential dependencies in a specification whose executability can allow system-level testing while avoiding the explosion of the number of use case sequences to generate? To answer positively this question we develop a specification for TOTEM's library system in Arnold and Corriveau's ACL specification language and explain how this specification can be used to generate a manageable number of tests. The key idea is that by modeling use case mutual independence we can considerably reduce the number of tests.

Acknowledgements

I wish to thank my supervisor, Professor Jean-Pierre Corriveau, for his help in suggesting a topic, guiding me through it and editing this dissertation. Without his help, advice and trust, my studies in this area would not have been possible. I feel very fortunate to have had the opportunity to work under the supervision of Dr. Corriveau. His immaculate vision paved the path for the smooth, timely, and successful completion of this thesis. I extremely appreciate his invaluable advice towards my future academic endeavors. I also want to thank my co-supervisor, Professor Wei Shi, for involving me in her Engage research project with Blueprint, as well as for her input and funding for this research. Her contribution to the generation algorithm I present was essential to the completion of this work.

I would like to thank Carleton University, School of Computer Science for giving me the opportunity to do my thesis work. I would like to express my appreciation to all office administrators at the School of Computer Science.

The constant prayers of my mother Hanifa Shaheen have enabled me to complete this strenuous journey. I also thank my wife Aysha Hina and my kids for their unconditional love and support. I appreciate their patience, kindness, care and valuable encouragement through the hard times.

Lastly, I would to thank all of my friends, colleagues and lab-mates for fruitful discussions and companionship throughout the years. I have been blessed with a friendly and cheerful group of fellow students.

List of Acronyms

ACL	Another Contract Language
CER	Contract Evaluation Report
eUC	Extended Use Case
IUT	Implementation Under Test
OCL	Object Constraint Language
OOSE	Object-Oriented Software Engineering
SUT	System Under Test
TRM	Testable Requirements Model
TTCN	Testing and Test Control Notation
UC	Use Case
UCM	Use Case Map
UCTS	Use Case Transition System
UML	Unified Modeling Language
UTP	UML Testing Profile
VF	Validation Framework

List of Figures

Table of Contents

1 Introduction

1.1 Context of Thesis

In his seminal book on Object-Oriented Software Engineering (OOSE), Ivar Jacobson (1992) introduced the notion of *use-cases* to capture the requirements of a software system. For Jacobson, a use case consists of a series of steps capturing the interactions between external actors and the system (viewed as a black-box). Each use case in fact typically captures several *scenarios* (Ryser and Glinz, 2000); each scenario corresponding to a possible path (or traversal) through the use case. Implicitly, use cases and their scenarios define the *responsibilities* (i.e., fine-grained functional requirements) of the system (Wirfs-Brock and MacKean, 2002). Some scenarios of a use case can capture basic (i.e., usual valid) use of the system, whereas others within the same use case can capture exceptions. For example, a use case for starting to use a bankcard at an ATM can include both i) a user using the correct PIN of a valid card on a working bank network and ii) several scenarios corresponding to exceptions (e.g., incorrect PIN, expired bankcard, unavailable bank network, out-of-service ATM).

The inclusion of use cases as one of the five views supported by the Unified Modeling Language (Pilone, 2013), the *de facto* standard for OO modeling, has led to their widespread adoption for OOSE. (For example Blueprint (2013), one of the leading tools for requirement engineering, is rooted in the notion of use cases.) While there has been much discussion on different *styles* for use cases (e.g., Wirfs-Brock, 2014) Jacobson's original conceptualization of a use case as a set of scenarios has endured and, moreover, served as the basis for scenario-driven methods for OO modeling as well as OO testing.

For example, Robert Binder (2000) has introduced the notion of *extended use cases* (eUCs) for system-level testing. Roughly put, an eUC can be thought of as a *parameterized use case*, the parameters of each use case corresponding to the path sensitization variables required to *cover* the different scenarios (i.e., paths) through that use case (*Ibid*.).

In UML, the requirements of a system are to be modeled using a set of use cases. These use cases, as well as their relationships (i.e., between themselves and with actors) are captured in a *use case diagram* (Pilone, 2013). Unfortunately, it is widely acknowledged that UML's support for modeling inter-use-case semantics is very limited. In fact, it consists of only two built-in stereotypes: i) a use case may include another one and ii) a use case may extend another one (*Ibid*.). The exact semantics of these relationships is still debated. More importantly, these two stereotypes are not sufficient to address (the possibly complex) sequential dependencies between use cases, as explained at length by Ryser and Glinz (2000).

Binder (2000) defines system testing as *concerned with testing an entire system based on its specifications.* This task involves the validation of both functional and non-functional (e.g., performance) requirements. In their seminal paper on system testing using UML, Briand and Labiche (2002) remark:

> *[T]hough system testing techniques are in principle implementation-independent, they depend on the notations used to represent the system specifications. In the context of UML-based object-oriented analysis, it is then necessary to develop techniques to derive system test requirements from analysis models such as use case models, interaction diagrams, or class diagrams.*

To this end, these authors propose the TOTEM methodology that addresses the generation of test requirements from a set of UML artifacts namely: a use case diagram, use case descriptions (in natural language), a sequence diagram for each use case, class diagrams,

and a data dictionary that describes each class, method and attribute. What sets TOTEM apart from most similar proposals (as will discussed in the next chapter) is that it does not focus only on use cases individually but first and foremost address the *sequential* dependencies between use cases. More specifically, TOTEM puts forth specific steps to i) represent such dependencies and ii) use these representations to generate corresponding *sequences of use cases* to test. However, as Nebut et al. (2006) point out, TOTEM's generation algorithm, which heavily relies on FUSION's (2014) notion of *interleaving*, presents one major problem: even for small examples, the number of sequences of use cases to test is essentially unmanageable (as discussed in the next chapter). It is this problem that this thesis addresses.

1.2 Research Question, Thesis and Methodology

Very few use case driven approaches to system testing consider sequential dependencies between use cases (Briand and Labiche, 2002). Of these few, most (e.g., (Ryser and Glinz, 2002, Nebut et al., 2006) rely on capturing sequential dependencies between use cases by means of some form of state machine (*Ibid*.). Such a strategy is problematic (see next chapter). In contrast, TOTEM proposes modeling sequential dependencies via a UML activity diagram from which a spanning tree is obtained and used for generation use case sequences to test. Without yet going into details, it is important to realize that such an activity diagram is essentially descriptive: UML semantics for use cases and their sequential dependencies are *not* verifiable against the execution of system under test (Binder, 2000, chapter 8). This observation is the starting point for the research question we ask in this work namely: *is it feasible to capture use cases and their sequential dependencies in a specification whose executability can allow system-level testing while avoiding the explosion of the number of use*

case sequences to generate? It is our thesis that the answer to this question is yes. In order to support this claim, we intend to tackle the same case study as the one presented in TOTEM, namely a simple library system. We first explain in the next chapter why the number of use case sequences to test explodes in TOTEM. We then develop, in chapter 3, an executable specification for this library system and explain how this specification can be used to generate a manageable number of tests by testing for use case independence. More specifically, we shall develop a model of the TOTEM library system in Another Contract Language (hereafter ACL). ACL is a high-level contract language that is closely tied to the concept of requirements. It was developed by Arnold and Corriveau (2010a, 2010b, 2010c) and offers several constructs, in particular scenarios and responsibilities, for system-level modeling. Most importantly, an ACL model is executable. Let us elaborate.

ACL makes use of *contracts* (Meyer, 1992) to represent the classes of an implementation under test (hereafter IUT). A contract consists of several different parts, which are illustrated in the next chapters. Contracts make use of both dynamic and static checks. Dynamic checks are performed by the *Validation Framework* (VF) run time and typically address behaviour, while static checks are done before execution of the IUT and look at a system's structure. More specifically, the VF executes the ACL model, requesting information from the executing IUT whenever needed. Given an ACL model is implementation independent, the VF requires one additional input (beyond the ACL model and an IUT) namely *bindings* that map:

1) each class of the IUT to the contract(s) it must obey
2) each responsibility of each contract to a procedure of the IUT

Bindings enable the executability of an ACL model: as the execution of an IUT starts, the VF starts the execution of the ACL model to which the IUT is bound. Each time an instance of a class is created, the VF creates a corresponding instance of the contract to which this class is bound. In ACL, a contract contains, among other components, responsibilities and scenarios. Each ACL scenario defines a grammar of valid sequences of responsibilities. Each time a procedure of the IUT is called, the VF executes the dynamic checks associated with the responsibility to which this procedure is bound. Such checks typically consist of assertions (e.g., pre- and post-conditions, and invariants, as in design by contract (Meyer, 1992)). While verifying such assertions, the VF is able to query the execution of the IUT at that specific point in time in order to obtain punctual values the VF uses for these checks and for possibly updating any variables of that specific contract instance.

Most importantly, dynamic checks are not limited to assertions and also involve scenario monitoring. Each scenario of a contract must define a *trigger* (in the form of a responsibility or an event, as illustrated in the next chapter). As a *contract instance* is executing, as soon as one of its scenarios is triggered, a corresponding *scenario instance* is created. Each contract instance must monitor its scenario instances. The key technical claim of this work is that if we model *both* the sequential dependencies and the mutual independence of use cases via scenarios in ACL, then we can avoid TOTEM's problematic explosion in the number of tests for these. The goal of the library case study presented in chapter 3 is to demonstrate this point while developing a model that is semantically as close as possible to the one found in TOTEM.

2 Related Work, Generative Programming and ACL

2.1 TOTEM

In this section, we review in detail the TOTEM (Briand and Labiche, 2002) methodology, which also allows us to introduce the TOTEM models that pertain to the library system we use as a case study in the next chapter. (All figures in this section come from the TOTEM paper.) The complete approach is summarized in Figure 1 below, which is a flowchart for the steps of the TOTEM method[1].

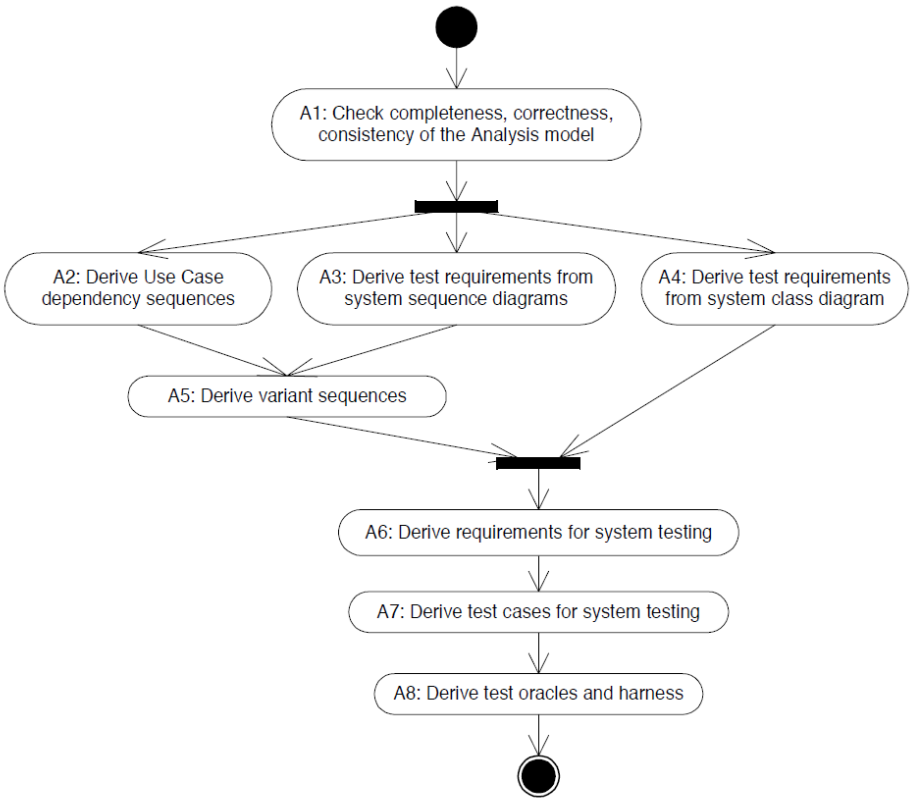

Figure 1: The TOTEM Methodology

[1] The notation for this flowchart is that of activity diagrams in UML.

However, the paper itself focuses on activities A2, A3, and A5. First, a textual description is developed for each use case of the system. Then sequential dependencies between these use cases are modeled using a UML activity diagram (see Figure 2).

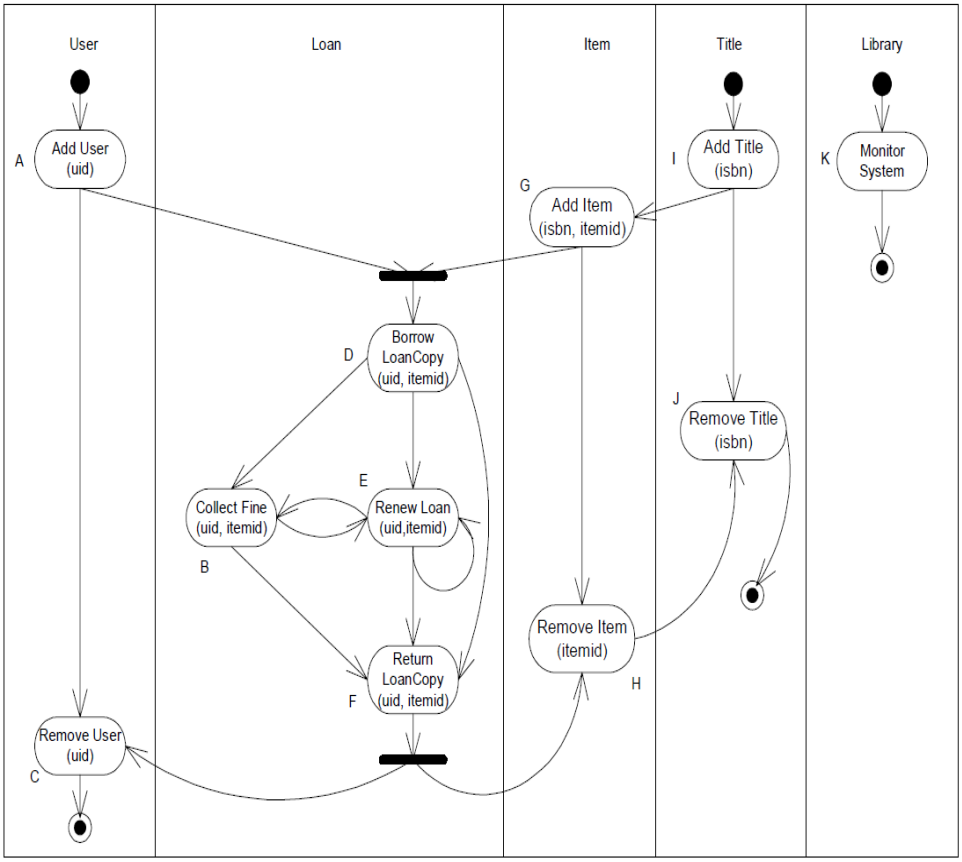

Figure 2. Use Case Sequential Dependencies

In this figure, each rounded rectangle corresponds to an extended use case (whose title is shown) and is given an alphabetical label. (The "Monitor System" use case is not used in TOTEM's case study.) The authors insist on the fact that they require extended use cases, that is parameterized use cases, in order to be able to distinguish the creation/deletion of a) different users (through a unique user id), different titles (via unique isbns), different copies (called items) of a same title (via a unique {isbn, itemid} pair) and different loans.

In Figure 2, the key point is that directed edges are sequential dependencies between use cases.

More precisely, the authors explain that:

> *"An edge between two use cases (from a tail use case to a head use case) specifies that the tail use case must be executed in order for the head use case to be executed, but the tail use case may be executed without any execution of the head use case. In addition, specific situations require that several use cases be executed independently (without any sequential dependencies between them) for another use case to be executed, or after the execution of this other use case. This is modeled by join and fork synchronization bars in the activity diagram, respectively."*

The use cases are grouped into swimlanes, according to their responsibilities in terms of manipulated objects. If not (even transitively) connected, use cases from different swimlanes can occur in any order. With respect to the total number of possible paths, Briand and Labiche remark:

> *The activity diagram in Figure 2 specifies an infinite number of paths, a property due to the loop between use cases CollectFine and RenewLoan. However, given that a loan can be renewed only twice, the number of paths between use cases BorrowLoanCopy and ReturnLoanCopy equals 14, thus leading to 130 paths in the whole activity diagram.*

These paths are determined by first transforming the activity diagram of Figure 2 into the graph of Figure 3 and then using a depth-first traversal of this graph (which is taken to account for loops and cycles) to obtain the spanning tree of Figure 4.

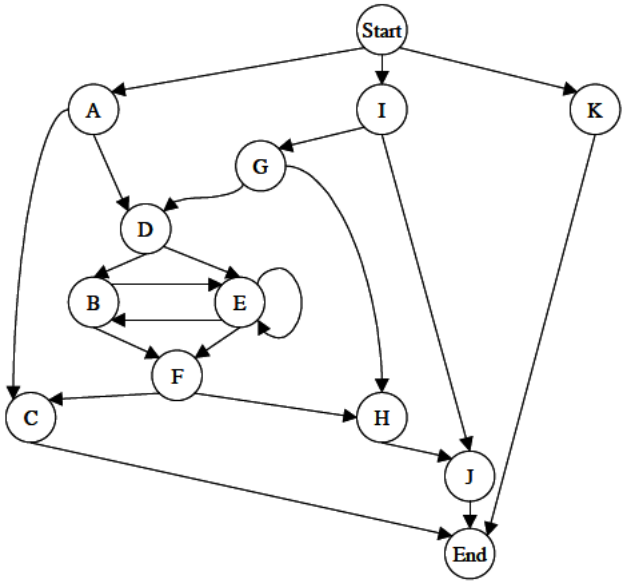

Figure 3. Directed Graph corresponding to previous activity diagram

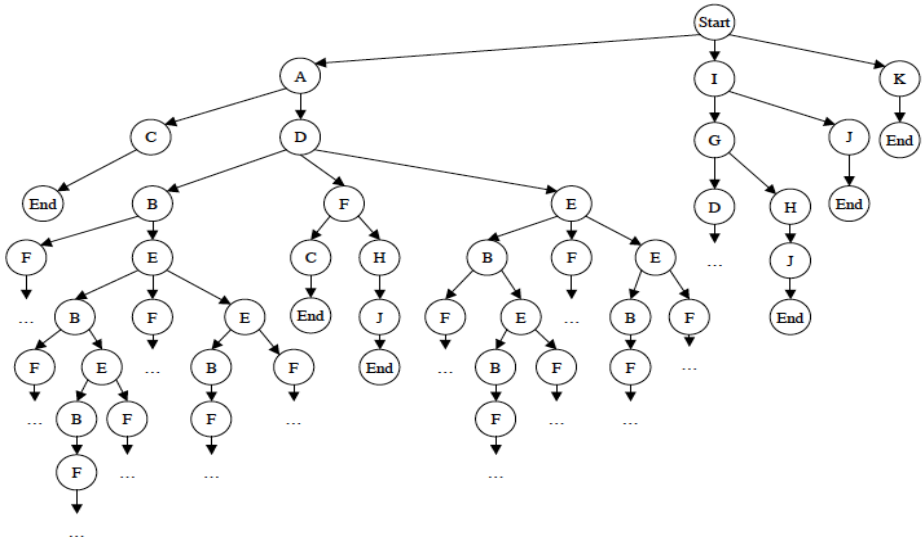

Figure 4. Tree derived from directed graph

Given such a tree, the authors next require what they call “test scale information”. Such information is crucial in order for their test generation algorithm to produce what they hope to be a manageable number of test cases. In their example, they assume “*that the tester*

wants: 2 users (u1, u2), 3 titles (t1, t2, t3), 2 items per title (e.g., i11 for copy 1 of title 1), 1 loan per user, no renew or collect fine for loans, and no system monitoring." They add: "*Such a situation, though not representative of a realistic use of the Library system, is deemed adequate as it implies a small number of sequences (7 out of 60 are now possible) but allows us to illustrate all the steps*." For this constrained set up, they then determine the regular expressions that capture the possible sequences of use cases. These are given below:

Parameterized Use Case Sequences	- `A(uid).C(uid)` - `I(title).J(title)` - `I(title).G(title, item).H(item).J(title)` - `(A(uid)||I(title).G(title,item)).D(uid,item).F(uid,item).(C(uid)||H(item).J(title))`
Parameter Instances (Symbolic values)	- Users (2): `u1, u2` - Titles (3): `t1, t2, t3` - Items (one per title): `(t1,i11), (t1,i12), (t2,i21), (t2,i22), (t3,i31), (t3,i32)` - Loans (one per user and item): pairs `(u1, i22)` and `(u2, i32)`

The last of these is key. Through the use of the interleaving operator (||), it models 9 possible sequences, (A || I.G) and (C || H.J) having 3 variants each. If we consider (A || I.G), this means that user creation can occur before, between or after the I.G sequence (i.e., Adding a title and then Adding a copy of that title). In TOTEM, all these possibilities must be tested.

Each parameterized use case sequence is then supplied in a specific way with actual values for its parameters. The details of this algorithm are not important here as Briand and Labiche confess that the latter is somewhat arbitrary. What matters is that a set of instantiated sequences is obtained:

Seq3: I(t2).G(t2, i21).H(i21).J(t2)

Seq4: I(t3).A(u2).G(t3,i32).D(u2,i32).F(u2,i32).C(u2).H(i32).J(t3)

Seq5: I(t1).G(t1, i11).H(i11).J(t1)

Seq6: I(t1).G(t1, i12).H(i12).J(t1)

Seq7: A(u1).I(t2).G(t2,i22).D(u1,i22).F(u1,i22).H(i22).C(u1).J(t2)

Seq8: I(t3).G(t3, i31).H(i31).J(t3)

The authors add: "Instantiated use case sequences are then combined together, in a stepwise manner, to produce *complete* use case sequences to be tested[.] Each time two sequences are combined, interleaving of instantiated use case subsequences can occur, thus possibly leading to large numbers of complete sequences to be tested". For example, Seq3 and Seq4 are combined and produce 495 interleavings. Once all these sequences have been combined, Briand and Labiche end up with 18018 interleavings! (The actual combination algorithms and computation of these numbers are given in that paper.) The key point to emphasize is that this is for a restricted context (see above) in which, in particular, there are few users, titles and copies and no fines or renewals. And there is no looping: the same copy of a title is never being borrowed twice over time by the same user, nor is it borrowed by two different users!!

It is this explosion in the number of test cases that is problematic, and it can be specifically rooted in TOTEM, like FUSION (2014), having to rely on interleaving.

Having discussed TOTEM's approach, let us now turn to other relevant approaches to system-level test generation.

2.2 Related Work

The idea of generating tests from specifications is not new. For example, Bernot (1991) presents a theoretical framework for testing programs against algebraic specifications (an idea that had been put forth decades earlier). In the same vein, Dick and Faivre (1993) extract a finite state machine from formal specifications. But such specifications often suffer from scalability issues (Coudert, 1993), especially those relying on theorem proving.

While there are many similar contributions (e.g., Tripathy and Naik, 1992; Carver and Tai, 1998), they are not directly related to our work because i) they are not necessarily targeted towards system-level testing and ii) they offer no semantics similar to UML's notion of use cases. Consequently, they are not discussed further in this thesis.

The widespread adoption of UML has led to several test generation techniques adapted to that particular notation. For example:

1) Basanieri et al. (2002) present a test generation technique that relies on the use of use cases and corresponding sequence diagrams. However, because the latter are anterior to UML 2.0, they are typically incomplete since each diagram can only capture one specific path of execution of the system. That is, there is no semantics for alternatives, concurrent and/or optional paths, nor for loops, among many omissions remedied in the sequence diagrams of UML 2.0. Consequently, there is a significant traceability gap between use cases and these inadequate (almost necessarily incomplete) sequence diagrams.
2) Offuut and Abdurazik (1999) have discussed how *transition-pair coverage* (see Binder, 2000) can be applied to UML statecharts. Binder also addresses this topic at length in chapter 5 of his book (*Ibid*.). However most such state-based approaches (e.g., Seifert et

al., 2003; Sokenou, 2006) are geared towards class testing (since this is the level of abstraction at which state machines are relevant in OOSE).

State-based approaches to system-level testing do exist. Consider, for example, Cockburn (1997), Fröhlich and Link (2000), and Ryser and Glinz (2000). All start with use cases in order to obtain a system-level statechart. That representation then lends itself to (not necessarily automated) test generation via various transition coverage techniques. These authors, however, readily acknowledge several difficulties with such a strategy. First, a major drawback of such work is its lack of automation: obtaining state-machines from use cases generally involves some manual process (akin to what Gomaa (2000) calls *consolidation*, that is, how state machines obtained for individual scenarios must be manually integrated together into a state machine for each class). Second, obtaining a state machine for a whole system immediately raises questions of scalability and traceability to requirements that have led to the abandonment of the structured analysis/structured design paradigm that rested on such problematic concepts.

These difficulties extend to more complex test generation techniques still rooted in the notion of a system-level state machine. For example, proceeding from the work of Fröhlich and Link (2000), Riebisch (et al., 2002) and Goëtze (2002) also transform use cases into a problematic system-level state machine from which they create a usage graph and a usage model. Despite its inherent flaws, the latter model serves as the foundations for a statistical approach to test case generation, a strategy that lies beyond the scope of this thesis.

In light of such difficulties, some researchers have put forth the idea of *generating* (through a technique called *state exploration*) a system-level state machine from a specification. UPPALL (Larsen et al., 1997), for example, is a state exploration toolbox that

uses timed automata for modeling, simulation and verification of real-time systems. As the main architect of Microsoft's state-of-the-art Spec Explorer (2013), Grieskamp's (2006) in-depth discussion of this technique is most informative. In that tool, modeling of requirements consists in defining an infinite transition system T_P through a Spec# program P (where Spec# is in fact a superset of C#). Exploration (which *attempts* to avoid state explosion through the use of complex heuristics and of action scenarios) reduces T_P to a finite test graph *G*. Test cases are subsequently generated by traversing *G* (using coverage techniques and/or user defined sequences of actions). Grieskamp makes it clear that the *state explosion problem* remains a daunting challenge for *all* state-based tools. Indeed, even the modeling of a simple game like Yahtzee can require a huge state space if the 13 rounds of the game are to be modeled. Spec Explorer offers a simple mechanism to constrain the state exploration algorithm by setting bounds (e.g., on the maximum number of states to consider, or the "look ahead depth"). But then the onus is on the user to fix such bounds through trial and error. And such constraining is likely to hinder the completeness of the generated tests, as Corriveau and Shi (2013) explain:

1) The generation of a system-level state machine without any constraints generally crashes due to state explosion.
2) Consequently generation is constrained by a 'scenario' that imposes restrictions on data and on valid sequences of activities. The result is referred to as a *sliced* state machine. It is important to remark that Spec Explorer typically do not correspond to a use case but at best to a specific path through a use case. The reason is simple: a use case generally has several paths and generally leads to state explosion.

3) Test generation applies to an individual sliced machine. Thus there is no mechanism to readily test sequential dependencies between the scenarios a same use case, let alone between use cases…

Yet, Ryser and Glinz (2000) eloquently demonstrate in their SCENT methodology how semantically rich temporal dependencies between use cases can be. The question then is whether some other (preferably UML-based) model can tackle such semantics.

Several researchers have suggested using UML's activity diagrams (instead of statecharts) for test generation. The idea is essentially the same as in TOTEM. For example, (Linzhang et al., 2004; Mingsong et al., 2006; Nanda et al., 2008; Biswal et al., 2008; Jena et al. 2014) all propose algorithms to transform an activity diagram into graph. Then, more or less complex algorithms (e.g., possibly involving genetic algorithms) are presented to generate (and possibly optimize) a test suite from such a graph. Unfortunately, these contributions generally do not address use cases, nor their dependencies. Instead, activities refer to methods of classes. In other words, these are class-testing approaches. Regardless, some, such as the one of Kundu and Samanta (2009), tackle synchronization faults between methods, a form of sequencing. However, the proposed solution rests on noticing changes in the state of objects, which is highly problematic. Hartmann et al. (2005) also use activity diagrams for system testing but do not address sequential dependencies between use cases. Instead, it is each use case that is converted into an activity diagram, which is then augmented with annotations pertaining to the category partitioning of the variables relevant to that use case. Ultimately, tests are generated for each use case independently of others. In contrast, both TOTEM and our work focus on tests pertaining specifically to use case sequential dependencies.

Further references to similar work on the use of UML models for test generation are available in several surveys including Khandai et al. (2011) and, especially, Shirole and Kumar (2013). It must be emphasized that several proposals start with one or more UML models but then augment these models with additional semantics. For example, Autili et al. (2006) suggest enhancing a subset of UML 2.0 Interaction Sequence Diagrams with property specification semantics (based on temporal logic) enabling the generation of Büchi automata. Beyond specific details, and while it is widely accepted that techniques for theorem proving and for state-based verification have greatly improved, the fact remains that the use of logic or state-based approaches does not readily scale up to system-level testing (Grieskamp, 2006).

Beyond UML, several scenario-based notations have been put forth for the purpose of verification and/or test generation. Several address concurrency and distribution, something neither TOTEM nor this work addresses. Some, like Petrinets (2013), have formal semantics that are very distant from use cases. Conversely, like UML models, others offer semantics that are not anchored in mathematics. In particular, Use Case Maps (UCMs) (Amyot and Mussbacher, 2011) are interesting because they are specifically meant to correspond to use cases. Test generation from individual UCMs has been explored (Amyot et al., 2005) but tests for sequential dependencies between UCMs have not. However, the availability of stubs[2] in UCMs does allow a UCM to capture such dependencies (much like in the activity diagram used in TOTEM, the activities are in fact use cases.) But the ability to model sequential dependencies between use cases does not entail the testability of this model, that is (according

[2] In a UCM, a stub (denoted by a diamond) is a placeholder for another UCM, which can be statically or dynamically selected.

to Binder (2000)), the ability to generate adequate test cases from this model. In particular, UCMs, like use cases in UML, are not parameterized. Yet, as TOTEM demonstrates eloquently, parameterization of use cases constitutes a crucial aspect of system-level test generation.

Consequently, one must acknowledge that the absence of parameters for use cases in UML entails that tools that do support UML 2.0 will not support test generation approaches that rely on something absent from the UML standard. In other words, there is currently no UML compliant tool that does system-level test generation to the level of detail advocated by TOTEM. Indeed, of the few system-level testing approaches rooted in use cases, TOTEM is the only one we know to illustrate how use case parameters are relevant as well to corresponding class diagrams (Appendix C), sequence diagrams (Appendix D) and relevant OCL constraints (Appendices E, F, and G). Indeed, parameterization is at the heart of TOTEM's approach to test generation:

a) Use case parameters are required on the regular expressions that TOTEM derives from the initial activity diagram and from which tests are generated.
b) TOTEM's use of interleaving also rests on use case parameters (as explained at the beginning of this chapter).

The point to be grasped is that without use case parameterization, system-level test generation is grossly simplified, regardless of the semantic nature and use of the underlying model. But it is equally important to recognize that use case parameterization directly (as in TOTEM) or indirectly contributes to the explosion of the number of test cases to generate. The work of Nebut et al. (2006) illustrates what we mean by indirect contribution. In this two-step process, one must first obtain test *objectives*, each capturing a correct sequence of use

cases. To do so, each use case has pre- and post-conditions that are used to *infer* the correct partial ordering of use cases. Then, one generates test scenarios from these test objectives. A test scenario takes the form of a valid sequence of *instantiated* use cases. The latter are use cases for which the *actual* values of their parameters are explicitly specified. Valid sequences are then captured using a state machine (called a use case transition system - UCTS) as the one shown in Figure 5 (*Ibid*.). This UCTS is then used to support a simulation of the system.

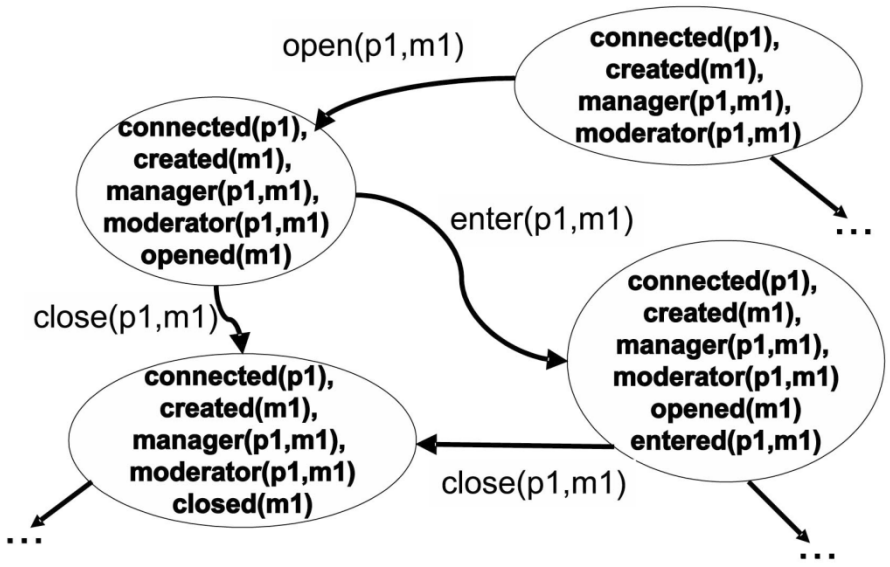

Figure 5. An Example of an UCTS

Here p1 and m1 are the ids respectively of a participant and of a meeting for the meeting service being modeled. Use cases *open*, *enter* and *close* all have two parameters: a participant and a meeting. Nebut et al. readily acknowledge that the size of an UCTS can be problematic. For example, they explain that an UCTS involving 2 participants and 2 meetings requires an UCTS of 1616 states! Clearly, the number of possible test scenarios to generate from such an UCTS explodes! In fact, the authors state that the scalability of their approach is *inversely* proportional to the number of instances required. In order to minimize this problem, they resort to a very weak excuse: “For the systems we studied, even if a large might be

present in the real system, only a small number of instances are necessary to achieve statement coverage and thus the size of UCTS remains acceptable". The flaw with such an argumentation is that statement coverage is too low a goal for system-level testing: it's not statements of a system under test that are to be covered but use cases *and* their sequencing!

The approach of Nebut et al. is further handicapped in relying, as TOTEM does, on interleaving semantics.

The importance of parameterization for modeling use case sequential dependencies leads to another important conclusion: UTP, the UML Test Profile (Baker et al., 2008) does not help with the abovementioned problems for two reasons:

a) It is a specialization of UML for the specific purpose of *describing* tests. Consequently, it does not address test generation per se.
b) While a sequence diagram (see Figure 6) can capture actual values for parameters of messages between objects, as previously mentioned, current UML semantics do not support parameterized use cases. Nor do UML 2's High Level Interaction Diagrams. Consequently, there is currently no mechanism to capture dependencies between parameterized use cases.

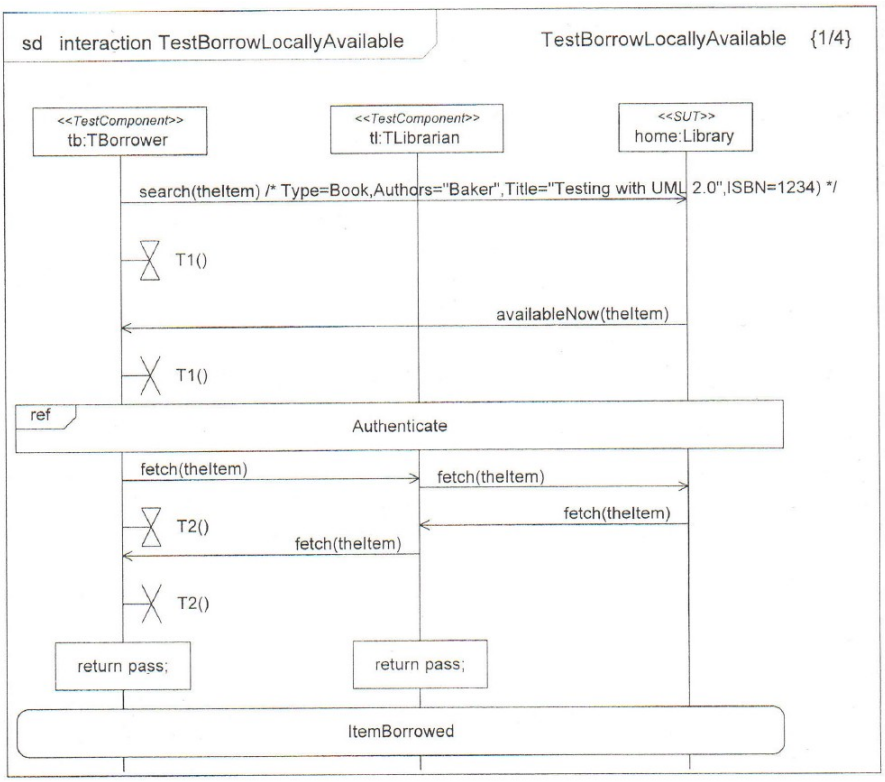

Figure 6: Test Case in UTP for Borrowing

Ultimately, this leads us to consider TTCN-3 (2014), a standard notation (offering several supporting toolsets). The Testing and Test Control Notation version 3 is a strongly typed test scripting language used in conformance testing. Contrary to most of the approaches discussed thus far, TTCN-3 offers rich semantics for describing data. In particular, it supports instantiated templates (such as *ParisWeekendBehaviorRequest* below) as well as parameterized templates (such as *myTypeB* below):

```
type record weatherRequest {
        charstring location,
        charstring date,
        charstring kind
}
```

```
template weatherRequest ParisWeekendWeatherRequest := {
        location := "Paris",
        date := "15/06/2006",
        kind := "actual"
}

type record location {
        charstring city,
        charstring country
}

template myTypeA myRomeLocation := {
        city := "Roma",
        country := "Italy"
}

template myTypeB myGenericWeatherRequest (myTypeA theLocationTemplate) := {
        location := theLocationTemplate,
        request := "road conditions"
}
```

Test cases can also be parameterized. For example:

```
testcase testWeather (myTypeB theWeatherRequest,
        someResponseType theExpectedResponse) runs on MTCType {
  weatherOffice.send(theWeatherRequest);
  alt {
        [] weatherOffice.receive(theExpectedResponse) {
                setverdict(pass)
                }
        [] weatherOffice.receive {
                setverdict(fail)
        }
  }
}
```

However, while TTCN-3 does support parameterization, one must acknowledge it is quite distant from the problem at hand for several reasons:

1) It is purely a notation for test cases and does not address per se test case generation.
2) The execution of TTCN-3 test cases requires the writing of complex adapters that are to bridge between the data descriptions in TTCN-3 and the actual types used in a system under test (SUT). Furthermore, the input and output of this SUT must

be organized in terms of ports used in the TTCN-3 specification. Consequently, there is a significant semantic gap between requirements as conceptualized as use cases and their capturing in TTCN-3.

3) TTCN-3 does not support the notion of a use case nor of a scenario. In fact, its approach to behavior specification essentially reduces to the creation of individual test cases, each consisting of nested if-then-else clauses called alternatives. Semantically, it is not equipped to capture sequential dependencies between use cases.

The solution we propose to capture such dependencies and generate tests for them rests on the use of a specification language to which we now turn.

2.3 Another Contract Language (ACL)

The specification language that is used for this thesis is called Another Contract Language (ACL). It was put forth by Arnold and Corriveau (2010a, 2010b, 2010c). In this section, we summarize the essential aspects of it through a simple example.

ACL scenarios are conceptualized as grammars of responsibilities. Each responsibility represents a simple action or task, such as the saving of a file, or the firing of an event. Intuitively, a responsibility is either bound to a procedure within an implementation under test (hereafter IUT), or to be decomposed into a sub-grammar of responsibilities. In addition to responsibilities and scenarios, the ACL offers a set of Design-by-Contract elements. They are typically used to express constraints on the state of the IUT before and after execution of a scenario or responsibility: Preconditions specify constraints on the state of

the IUT before a responsibility or scenario can be executed. Post-conditions specify constraints on the IUT's state following a successful responsibility or scenario execution. The ACL also provides the means to express invariants.

When a pre- or post-condition fails, execution proceeds but the failure is logged in the Contract Evaluation Report (CER). Also, when a scenario is executed by an IUT, the specified grammar of responsibilities must hold. That is, the responsibilities that compose the scenario must be executed in such 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 IUT does not match the Testable Requirements Model (hereafter TRM). This mismatch is also reported in the CER.

The following annotated example summarizes several of the semantics currently supported by our contract language and VF. We have included example usage of inheritance in order to demonstrate how our work supports the compositionality of contracts through this mechanism. However, inheritance as well as some other features of ACL are not used in our case study.

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.

```
Import Core;
Namespace My.Examples
{

abstract Contract ContainerBase<Type T>
{
  Scalar Integer size;
}
```

An observability is a query-method (like a get method in object-oriented programming) that is used to provide state information about the IUT. That is, they are methods that acquire and return a value stored by the IUT. An abstract observability MUST be refined in a derived contract.

```
Observability Boolean          IsFull();
Observability Boolean          IsEmpty();
Observability T ItemAt(Integer index);
Observability Integer   Size();
abstract Observability Boolean  HasItem(T aItem);
```

The body of the "new" responsibility is executed immediately following the creation of a new contract instance. Similarly, the body of the "finalize" responsibility is executed immediately before the destruction of the current contract instance.

```
Responsibility new()
{ size = 0;   Post(IsEmpty() == true); }

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()); }
```

The following 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. Index is not related to size. Responsibility GenericAddition() takes care of increasing the size.

```
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)));  }

}
```

A testable requirements model (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;
Scalar Timer item_timer;
Scalar Integer number_of_items;
Parameters
{ Scalar Boolean CheckMembers; }
```

The abstract responsibility of ContainerBase is now refined. Value is the keyword for return value.

```
refine Observability Boolean HasItem(tItem item)
{ tItem x;  Boolean result = false;
  loop(0 to Size())
          { x = ItemAt(counter);
            result = result || x == item;       }
  value = result;.          }
```

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
    { Belief CheckMember("There should be a member in our container to hold elements of type
tItem")
          { HasMemberOfType(tItem); }    }    }

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);
  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 by Arnold and Corriveau (SERA 2010b)

```
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()); }


Metric List Integer ContainerTimes()
{ context.container_times; }

Metric Scalar Integer NumberOfItems()
{ context.number_of_items; }
```

This section builds the evaluation report. {0} is where the reported result goes in the output string. A report all statement performs the exact same way as the report statement, except that it generates a single result for all contract instances.

```
Reports

{ Report("The average time in the container is {0} milliseconds",   AvgMetric(ContainerTimes()));

  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());
          }

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

2.4 Running an ACL Contract in the Validation Framework

ACL supports automated validation. That is, once a TRM is linked to an IUT, all checks are automatically instrumented in the IUT whose execution is also controlled by the VF (e.g., in order to monitor scenario instance creation and execution). Thus, our whole approach to validation hinges on the ability to link a TRM to an IUT. To do so involves the creation of *bindings*. That is, the binding process also allows a) responsibilities be mapped to (possibly sequences of) procedures of the IUT and b) contracts to set variable types and return types at the time of binding.

After a contract has been successfully bound to an IUT, a report can be generated. This report shows how well the IUT conforms to the contract. For an example of the layout of the report, please refer to Figures 7 through 9. Figure 7 shows an example in which one of the scenarios failed. The part that failed is shown in red on the left. In the overall report, the result is listed as “Fail”, and it is shown that one scenario and one contract failed. This contract failed because the scenario inside it failed. In Figure 8, the reason the scenario failed is shown. In this case, it is because the scenario failed to run correctly. The report seen in Figure 9 succeeded and, therefore, in the overall result it is declared as having passed, and everything on the left is green.

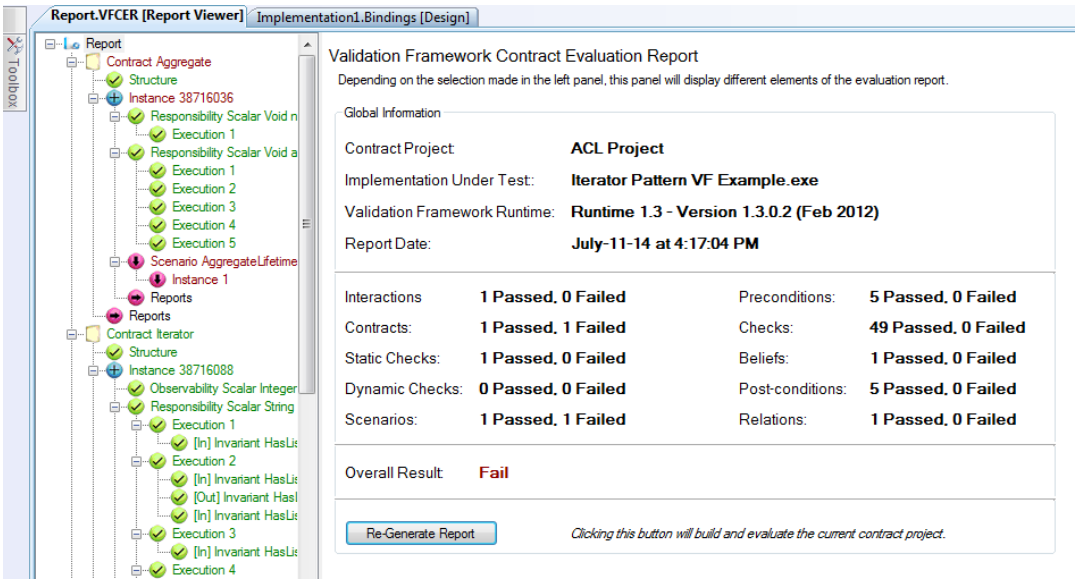

Figure 7. VF Report - Failure

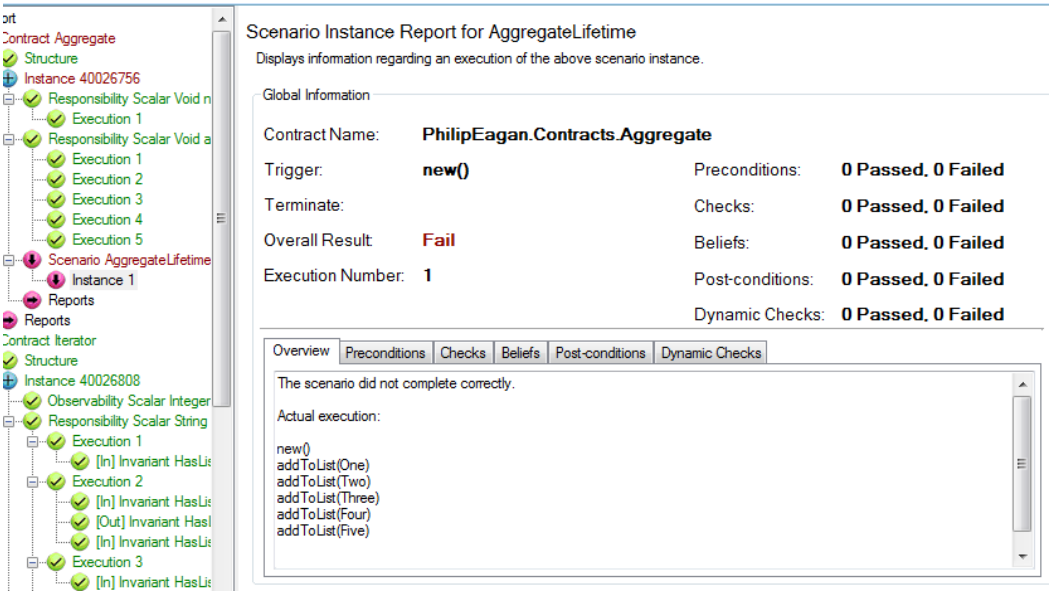

Figure 8. VF Report - Failure Drill Down

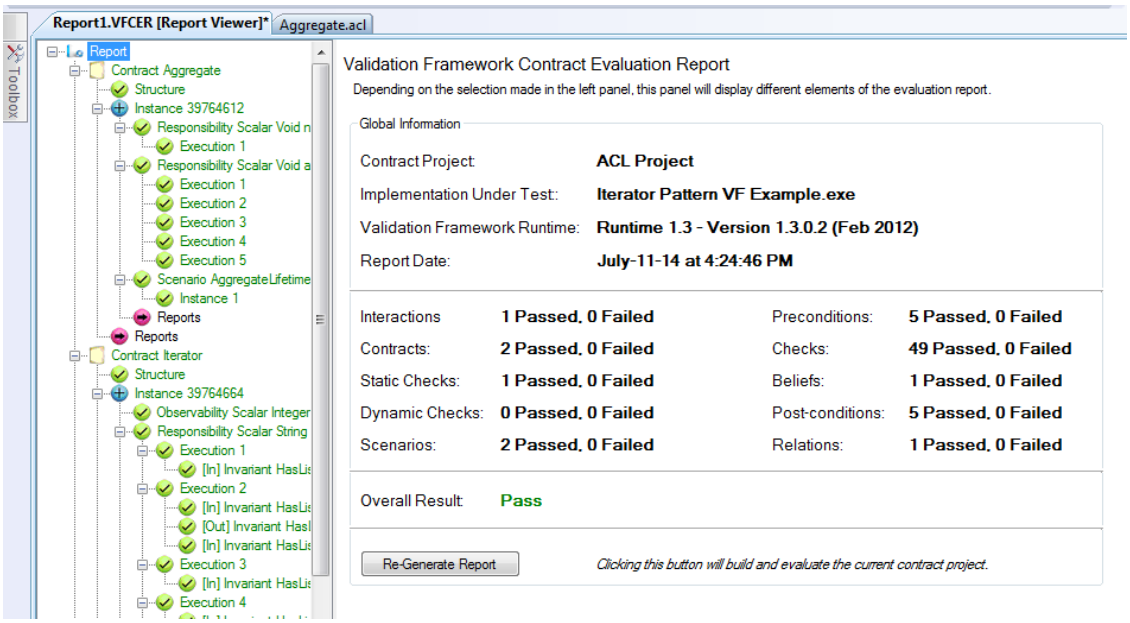

Figure 9. VF Report - Pass

Having overviewed ACL and the VF, we now turn to the modeling of the Library System of TOTEM using ACL.

3 Comparison of Approaches

3.1 Overview of the Library System

In the rest of this thesis we focus specifically on the library system used as the case study in TOTEM. In this subsection, we summarize the basic characteristics of this system, which are taken from TOTEM (Briand and Labiche, 2002), a paper we will simply refer to hereafter as TOTEM. The library system is required to manage various functions. It is designed to have a large number of users and titles, each with a unique id. Each title is identified by a unique ISBN and is associated with other information (such as author(s), publishers, etc.). Each title can have several copies, each with a unique copy number. We will adopt TOTEM terminology and refer to such copies as items. A specific book can therefore be identified by a unique pair {isbn, copy number}.

We will limit ourselves to considering most of the functionality TOTEM defines for its library system. (As in TOTEM, we specifically exclude the monitoring aspects that are irrelevant, as they have no interaction with the rest of the system.) TOTEM postulates that a single librarian (who, for example, accepts transactions from users for issuing, returning and renewing books) drives the system (which, therefore, does not offer direct access to users). The librarian acts as an administrator and controls the system through the interface of a terminal. We do not focus here on any specific interface.

A librarian can create or delete users, titles and items. Users can borrow, via the librarian, an available item, as long as they are registered and do not have late fees associated with their account. Specifically, a user will accrue fees if s/he does not return a book within a stipulated timeframe. Users are allowed to borrow a book, as long as they have no overdue loans or have not reached the maximum allowed number of books to borrow. One important

detail: despite having a user id and an item id as parameter, the “Collect Fine” use case of TOTEM states that a user must pay the entire fine accumulated to clear a late fee. That is, a user cannot pay only a portion of a fine: only complete payment is accepted.

Based on this description, we now turn to our ACL model proper.

3.2 ACL Model

In order to keep our model as similar as possible to the one of TOTEM, we have it consist of a single contract for the librarian. Clearly, if we were to deviate from the TOTEM model, we could have instead used several contracts. Such a more decentralized organization of the model would definitely complicate the syntax required for modeling, as well as the algorithm for generating test paths. But it would not necessarily entail an increase in the number of paths generated in comparison to using a single contract. Consequently, given our goal is to demonstrate it is feasible to model in ACL a system identical to the one of TOTEM but reduce the number of paths that need to be generated from this system (especially with respect to loops and interleaving), we will keep in this thesis to a ‘centralized’ model with a single contract to achieve this goal, leaving the multi-contract model as future work.

We will introduce this contract using a use-case perspective. That is, we will present and explain the portion of the contract that pertains to one use case before moving to the ACL that proceeds from another use case. We follow the order of presentation of use cases found in TOTEM. We therefore start with the setting up of the contract and the use case for user addition (i.e., creation). Throughout this discussion, we will not focus on the syntactic details of ACL (which are explained in ACL’s online specification) but rather on semantics.

```
Import Core;
Namespace ACLLibraryContracts.Contracts
{
 Contract Librarian
 {
   Parameters
    {
          [0-10] Scalar Integer MaxLoanCount;
          [0-2] Scalar Integer MaxLoanRenewalCount;
          [0-3] Scalar Integer MaxLoanInterval;
     }

 // contract variables
String potentialNewUserInfo = null;
Integer userIdToBeDeleted=0;

/*~~~Contract variables for testing the immutability of relevant items ~~~*/

Dictionary(Integer, Integer)  statesOfUsers;    // key: a user id, value: a state signature
Dictionary(String, Integer)    statesOfTitles;    // key: a title string, value: a state signature
Dictionary(Integer, Integer)  statesOfItems;   // key: an item id, value: a state signature
Dictionary(Integer, Integer)  statesOfLoans;     // key: an loan id, value: a state signature


/*~~~Creating a user~~~*/
Observability Boolean DoesUserExist(String userInfo);

/*~~~Observabilities for testing the immutability of relevant items~~~*/

Observability Integer getUserState (Integer uid);
//each userId uniquely identifies a user
Observability Dictionary(Integer, Integer) getUsersStates ();

Observability Integer getTitleState (String aTitle);
Observability Dictionary(String, Integer) getTitlesStates ();
// alternatively we could have built a dictionary using isbn as key

Observability Integer getItemState (Integer itemId);
Observability Dictionary(Integer, Integer) getItemsStates ();

Observability Dictionary(Integer, Integer) getLoansStates ();
```

```
Responsibility String  initiateNewUserAddition()
{
        Pre(potentialNewUserInfo ==  null);

        // gather system state BEFORE user creation
        statesOfUsers = getUsersStates();
        statesOfTitles = getTitlesStates();
        statesOfItems = getItemsStates();
        statesOfLoans = getLoansStates();


        Execute();
        potentialNewUserInfo = value;
        Belief userCreationStarted("Librarian has started the user creation through
                                    the Library Terminal");
        Belief librarianEntersUserInfo("Librarian was prompted to enter the user
                                        info through the Library Terminal");
        Post(potentialNewUserInfo not= null);
        fire(userAdditionRequested(potentialNewUserInfo));
}

Responsibility Integer generateNewUserId(String userInfo)
{
        Pre (userInfo not = null);
        Execute();
        Belief ProperUserIdGenerated("A unique userId is generated
                                                for the given userInfo");
        Post(value >0);
}

Responsibility createUser(String userInfo, Integer uId)
{
        Pre(userInfo not= null);
        Pre(uId >0);
        Execute();
        Post(DoesUserExist(userInfo) == true);
        fire(newUserCreated(uId));
}
```

```
Scenario addUser
{
   Integer uId;
   String newUInfo;
   Trigger(observe(userAdditionRequested(newUInfo)));
   choice(DoesUserExist(newUInfo))  true
  {
         Belief reportUserExists("UI should inform the librarian that user
                                          already exists with the given information");

  }
   alternative(false)
  {
         uId = generateNewUserId(newUInfo);
         observe(newUserCreated(uId));

         // check user creation has not modified incorrectly previous state of the system
         statesOfUsers.AddPair(uid, getUserState(uId));
         choice (  (statesOfUsers != getUsersStates())  |
                   (statesOfTitles != getTitlesStates())    |
                   (statesOfItems != getItemsStates())    |
                   (statesOfLoans != getLoansStates())  )  true
         {
            Error("UI should inform the librarian that the
                   System has incorrectly modified objects that should have not been
                  modified.");
         }

         fire(userAdditionCompleted(uId));
  }
  potentialNewUserInfo = null;
  Terminate();
}
/*~~~End Creating a user~~~*/
```

The ***Librarian*** contract begins with an ***import*** statement, which is used to reference the plug-in namespace that will be used within the contract. The contract begins with the use of the ***Contract*** keyword, which is followed by an identifier denoting the name of the contract. (The contract name is also used as the binding point to reference when a binding is made between the contract and a type defined within the IUT.) Once a contract is defined, the body is specified between opening and closing brace brackets (‘{‘ and ‘}’).

A number of different sections exist in the body of the contract. In our ***Librarian*** contract, the first section found is a ***parameters*** section. Parameters are used to configure the contract and can also be used to specify different scenario paths, or simply to denote a constant value, such as ***MaxLoanCount***.

For this use case there are three parameters. ***MaxLoanCount*** is of type Integer and is a Scalar (as opposed to a List). The declaration of ***MaxLoanCount*** starts with the specification of a range of possible values. (If no range information is specified, then the parameter can be assigned any value, constrained to the parameter type.) The range for ***MaxLoanCount*** indicates acceptable values are between 0 and 10. This parameter stores the maximum number of concurrent loans for a single user.

The parameter ***MaxLoanRenewalCount*** is used to store the maximum allowed number of renewals and ***MaxLoanInterval*** is used to store the maximum allowed number of days for which a loan can be held. If a loan is not returned within this stipulated period, then a user's privilege (i.e. ability) to borrow will be revoked and a fee imposed (to be paid in full to regain the privilege to borrow).

Following the parameter section we find the definition of six contract variables, whose scope is the whole contract. They are used to store information used during the evaluation of the contract. The ***potentialNewUserInfo*** contract variable is a string initialized to null and the ***userIdToBeDeleted*** contract variable is initialized to zero. The next four contract variables are of type dictionary, which is typical dictionary data-type to store key-value pairs. The purpose of these contract variables will be explained shortly.

Next, we declare the observability ***DoesUserExist()*** that is to be bound to an IUT method that will return ***true*** if there exists a user for the given user information; ***false***

otherwise. That is, once this contract has been supplied with bindings to the IUT at hand and compiled, it will be evaluated against an execution of this IUT. During this process, each time this observability is referred to, the corresponding method of the IUT will be called (with the relevant parameter) and will return a Boolean value that the contract evaluation process will use.

Next, we declare the observability ***getUserState()***, which is to be bound to an IUT method that will return, for a specific user, an integer value, representing its state signature. The idea of a state signature is a common industrial technique used to verify state immutability. It consists in computing a 'checksum' of the values of the instance variables (i.e., data attributes) of a specific instance. The method bound to the observability ***getUserState()*** returns this state-signature for the user object that has the userId equals to the userId passed through the observability. While state signatures are defeasible, such rare occurrences are not considered in this thesis. We acknowledge this is a simplification that needs to be addressed in future work. Thus we assume that if the state of an object changes then so does its state signature.

We then declare an observability ***getUsersStates()***, which returns a dictionary with user ids as keys and state signatures as values. Similar observabilities for titles, items and loans are also declared. How to use all these observabilities will be illustrated shortly.

In our ***Librarian*** contract, the use case "Add User" involves three responsibilities:

The first is ***initiateNewUserAddition()***, which is to be bound to a method that accepts the user information entered by the librarian through the library terminal. The precondition ***(potentialNewUserInfo == null)*** checks that variable ***potentialNewUserInfo*** is or has been reset to null. That is, any previous user creation must reset this variable to null once that

addition is completed. We then use the abovementioned state getter observabilities in order to obtain a snapshot of the state of all objects whose state must not change during user creation.

Then the keyword ***Execute*** is used to indicate the bound method is to be executed. The return value of this method is stored in ***potentialNewUserInfo*** (via the keyword ***value***). In essence, this return value is taken to be a serialized version of the information provided by the librarian for the creation of the user at hand. How this is achieved in the IUT is of no concern to the ACL model. What matters is that we use ***potentialNewUserInfo*** to properly track and differentiate between each user-creation request. Consequently, the model must be set up so that this variable is indeed reset to null at the end of this attempt to create a user (be it successful or not).

Two ***Belief*** statements follow. A ***Belief*** is used to remind the tester to observe something in the execution that the VF cannot verify. Here, the first belief corresponds to the requirement that there must be a way for the librarian to initiate user creation. The second belief corresponds to the requirement that the librarian has to have submitted information about the new user. The post-condition then ensures that the system indeed got input and that it was captured in ***potentialNewUserInfo***.

Finally, we have a ***fire*** statement used to let the model know that a new user creation request has been initiated. Most importantly, this event has ***potentialNewUserInfo*** as a parameter. Such a parameterized event allows us to differentiate between distinct user creation requests, as will be explained shortly.

The second responsibility ***generateNewUserId()*** is bound to a method that, given serialized user information, generates a unique positive user Id. The precondition ensures the

parameter is not null. Then, after the ***Execute***, the ***Belief*** statement captures the fact that this ACL model, as the TOTEM one, does NOT verify whether the id returned by the method bound to this procedure is unique. It should be emphasized however that, contrary to TOTEM, our ACL model could easily verify the uniqueness of such ids. It would merely require another contract variable to keep the list of current ids. Then it would be trivial to test if the value returned by the method bound to this responsibility is indeed unique. It must be emphasized that it is the ability of ACL to obtain at run-time values from a running execution that allows such checks, which most system-level testing approaches simply do not offer. Here, given uniqueness checking is left to the tester of the IUT proper to verify, the post-condition merely verifies that the return value is indeed an id (i.e., a positive integer).

The third responsibility, ***createUser()*** is meant to address completing user creation. To do so, it requires two parameters: the serialized information of a user and that user's id. It first verifies both have valid values, then executes the method bound to it. It then uses observability ***DoesUserExist*** to have the IUT assert that indeed the user has been created. At that point, an event broadcasts this fact to the rest of the contract. And there lies a key point to grasp: these three responsibilities correspond to the steps of the use case. But the model has not yet connected them. This is the purpose of the ***addUser*** scenario that follows these responsibilities: It is this scenario that corresponds to the 'Add User' use case given in Appendix B of TOTEM. However, it is important to remark that there is a noticeable difference: whereas TOTEM's use cases are parameterized, *current* ACL syntax does not allow scenarios to have parameters. But, semantically, TOTEM's parameters for this scenario can be mimicked using variables as explained shortly.

The ***addUser*** scenario starts by declaring two local variables: one to hold a user id, and one to hold the serialized information of a user. A scenario must be triggered. This particular scenario is triggered each time the event ***userAdditionRequested*** is fired. That is, using the keyword **observe** within a **Trigger** statement in ACL ensures that this scenario will be notified each time the event, specified as a parameter of the **observe**, is fired. Upon each such notification (from a **fire** statement), ***newUInfo*** is set to the value of the parameter of that event in the notifying ***fire*** statement. The event ***userAdditionRequested*** is fired in responsibility ***initiateNewUserAddition()***. Thus a *scenario instance* is created every time a new user addition is requested. It is important to notice that such a request could involve attempting to create a user who already exists (as mentioned in the corresponding TOTEM use case). This explains the rest of the scenario: Following the trigger statement, the ***choice*** statement uses observability ***DoesUserExist*** to check whether the supplied user information corresponds to an already-created user or not. If it does, a ***Belief*** statement is used to capture that it is up to the system to make sure the user interface informs the librarian of this attempted redundant creation. Alternatively, if this indeed a new user, then the ***alternative*** branch of the ***choice*** statement will be used to continue with the other steps of this valid creation. First, for the execution of this scenario instance to succeed, the ***generateNewUserId*** responsibility must be invoked to obtain a unique id for this new user. Then an ***observe*** statement will be used make sure that responsibility ***createUser()*** was invoked and did complete correctly by firing the ***newUserCreated*** event with the correct user id. Thus, it is in the scenario that valid sequences of responsibilities are captured. Next, the contract variable ***statesOfUsers,*** is appended with a key-value pair having the userId of the user just created as key and state of that new user as value. Then a choice statement uses observabilities

getUsersStates(), getTitlesStates(), getItemsStates() and getLoansStates to fetch the current state of all objects whose states should correspond to those stored in the corresponding contract variables. The statement ensures no incorrect state change has occurred. An error statement is used to **stop** the execution of the model if an incorrect state change has occurred.[3] Clearly, such an approach rests not only on the use of state signatures but also on the fact that there is no concurrency in the modeled system. How ACL could tackle concurrency and distribution is an open problem.

It must be emphasized that it is not sufficient to only verify the state of users: the states titles, items and loans are considered to address the possibility of incorrect dependencies. It must also be noted that observabilities do not presuppose nor impose any particular type of representation at the IUT level for users, titles, items and loans (and their states). But because these 4 types of objects are relevant at the system-level, the model does impose that the IUT be able to report on these objects and their states, independently of their internal representations. Ultimately, the strategy of taking a snapshot of relevant states, updating the relevant state in light of the execution of a specific use case, and then checking that all states expected to remain unchanged indeed have allows us to test for a use case being independent from another. This strategy is reused throughout the model.

Finally, the scenario fires the event ***userAdditionCompleted*** with this user id (which is viewed by TOTEM as the sole parameter of this use case) in order for other scenarios to deal

---

[3] Please note that the Error statement does not have the same syntax as the Belief statement: it does not name the error but rather relies on a message to the user before stopping execution.

with the sequential dependencies of this use case. (An empty ***Terminate()*** indicates that the scenario automatically terminates when execution reaches the terminate statement.)

The scenario concludes with having the variable ***potentialNewUserInfo*** reset to null. Recall the latter is a contract variable that must be null in order for the responsibility ***initiateNewUserAddition()*** to satisfy its precondition. It is this same responsibility that sets the value of that variable. Waiting until the end of scenario ***addUser*** to reset this variable ensures two instances of this scenario cannot overlap in time. That is, whether trying to create a new user or a user that already exists, the execution of the current scenario instance must complete before a new attempt at creating a user is possible. It must be emphasized that TOTEM's specification of use cases is so abstract that it simply does not consider this problem. In fact, very few use case modeling approaches consider inter-use-case dependencies (see previous chapter) and none, to the best of our knowledge, address sequential dependencies between instances of a *same* use case.

From a semantic viewpoint, this scenario in fact tackles two sequential relationships: a) the possible paths through the steps of the corresponding use case and b) the sequential non-overlapping between its instances. In contrast, we will see later that there are typically several sequentially overlapping instances of the scenario associated with loans. We will also discuss shortly how our model addresses the sequential dependencies between this use case and others. To do so we must first deal with user deletion. Here is the ACL for that aspect of the model:

```
/*~~~Deleting a user~~~*/

Observability Boolean UserHasLoan(Integer userId);
Observability Boolean UserHasPrivilege(Integer userId);
Observability String  FindUserInfo(Integer userId);

Responsibility initiateNewUserDeletion(Integer userId)
{       Pre(userIdToBeDeleted == 0);
        Pre(userId >0);
        userIdToBeDeleted = userId;

        // gather system state BEFORE user creation
        statesOfUsers = getUsersStates();
        statesOfTitles = getTitlesStates();
        statesOfItems = getItemsStates();
        statesOfLoans = getLoansStates();
        Execute();
        fire(userDeletionRequested(userIdToBeDeleted));
}

Responsibility deleteUser(Integer userId)
{       Pre(userId >0);
        Execute();
        Post(DoesUserExist(userId) == null);
        fire(userDeleted(userId));
}

Scenario removeUser
{
String uInfo;
Integer newUIdToBeDeleted;
Trigger(observe(userDeletionRequested(newUIdToBeDeleted)));
uInfo = FindUserInfo(newUIdToBeDeleted);
choice (uInfo not=  null) true
 {  Belief displayUserInfo("User's Info should be displayed on library
                                    terminal");
   choice (UserHasPrivilege(userIdToBeDeleted)) true
   {
     choice (UserHasLoan(newUIdToBeDeleted))false
     {    deleteUser(newUIdToBeDeleted);
          observe(userDeleted(newUIdToBeDeleted));
          // check user deletion has NOT altered previous state of the system
          statesOfUsers.RemovePair(newUIdToBeDeleted,
                                                  getUserState(newUIdToBeDeleted));
         choice (  (statesOfUsers != getUsersStates())  |
                   (statesOfTitles != getTitlesStates())    |
                   (statesOfItems != getItemsStates())     |
                   (statesOfLoans != getLoansStates())    )  true
                  {
```

```
                    Error("UI should inform the librarian that the
                     System has incorrectly modified objects that should have not been
                    modified.");
                    }
            fire(userDeletionCompleted(newUIdToBeDeleted));
        }
      alternative (true)
      {
        Belief informUserHasLoan("librarian should be informed that
            user can't be deleted because user's has loan");
       }
   }
alternative (false)
 {
          Belief informUserDoesn'tHavePrivilege("librarian should be
          informed that user can't be deleted because user's
          privilege is revoked");
   }
alternative (false)
{
  Belief informUserDoesntExist("librarian should be informed that   user
                    doesn't exist by a message displayed on library terminal");
}
userIdToBeDeleted = 0;
Terminate();
}

Scenario userDeletionFollowsUserCreation
{
          Integer newUId;
          Trigger(observe(userAdditionCompleted(newUId));
          Terminate(observe(userDeletionCompleted(newUId));
}
/*~~~End Deleting a user~~~*/
```

Modeling TOTEM's 'Remove User' use case by itself is rather straightforward. We first define three observabilities. Observability ***UserHasLoan()*** returns **true,** if the user with the given userId (supplied as parameter) has any active loans; otherwise, it returns **false**. Observability ***UserHasPrivilege()*** returns **true** if the user with the given userId has borrowing privilege; otherwise, it returns **false**. A privileged user can renew a loan and borrow more

items (while obeying the rules stated earlier). Observability ***FindUserInfo()*** returns user information (as a string) if the user with the given userId exists; otherwise, it returns null.

This use case involves two steps, each model by a responsibility. First, ***initiateNewUserDeletion()*** has the method bound to it handle the incoming user deletion request. Given a user id, it first checks contract variable ***userIdToBeDeleted*** is 0, which is used to make sure there is no attempt at concurrent user deletions. That is, as for user creations, a contract variable is used to ensure the last attempt at a deletion completed before a new one is attempted. Next we verify the validity of the parameter value and store it in the contract variable ***userIdToBeDeleted***. Next, as for user creation, relevant state information is gathered. After the execution of the bound method, we fire event ***userDeletionRequested*** with the correct user id. The second responsibility deals with making sure that, upon the deletion of a valid user, that user is indeed removed, as verified in the post-condition. The purpose of firing the ***userDeleted*** event will be explained shortly. The complexity of the use case rests in scenario ***removeUser*** that corresponds to the use case of the same name given in TOTEM.

The scenario is triggered each time the ***userDeletionRequested*** event is fired. As with the previous scenario, redundant requests are handled. First, the observability ***FindUserInfo ()*** is used to find the user information corresponding to the supplied user id. This information is stored in local variable ***uInfo.*** The ***choice*** statement verifies whether this information is null or not. If it is, the last ***alternative*** at the end of the scenario uses a ***Belief*** statement to deal with a non-existent user. If the user exists, a ***Belief*** statement states that that user's information must be displayed on the terminal. Then the next ***choice*** checks whether this user has borrowing privileges. If this is not the case, then an alternative branch of the scenario uses a belief to require that the librarian be informed about the revocation of this user's privileges.

Should the user have borrowing privileges, then a last ***choice*** checks if that user has any outstanding loans. If this is the case, then an alternative branch requires again the librarian to be informed of this situation. Otherwise, the scenario requires that the responsibility ***deleteUser()*** be invoked with the correct user id. Then the event ***userDeleted*** must be observed in order to confirm ***deleteUser()*** completed successfully. Next, the key-value pair for the user being deleted is removed from statesOfUsers and a choice point is used to verify no incorrect state change has occurred. Only in the case of a successful deletion will the firing of event ***userDeletionCompleted*** will occur, thus enabling the verification of the ordering of the two use cases (as seen shortly). That is, verifying the sequential ordering of the two use cases only pertains to the valid paths of these use cases. Let us elaborate on this crucial point. Use cases capture several paths (also called scenarios, not to be confused with ACL scenarios that, in fact, correspond to use cases). Some of these paths denote valid executions whereas some correspond to exceptions. However sequential relationships between use cases pertain *only* to the successful paths of the ordered use cases. In other words, semantically, attempting to sequentially order an unsuccessful addition with an unsuccessful deletion (of a user, of a title, etc.) does not make sense. Most of the modeling of sequential relationships in our model proceeds from this postulate.

For example, the scenario ***userDeletionFollowsUserCreation*** is used to enforce the ordering (of a successful user addition with a successful user deletion) in an extremely simple way: it is triggered by the successful completion of the creation of a new user and it terminates upon the successful completion of the deletion of the same user. The simplicity of this scenario proceeds from the use of parameterized events specifically associated with successful addition and successful deletion of a *same* user (captured via the parameter). That

is, we repeat, each use case captures several paths including exception ones (such as attempting to create an already-existing user, attempting to delete a non-existent user, etc.). Every TOTEM use case can be captured in an ACL scenario of a contract. By firing events associated with specific paths in distinct use cases, it is possible to define a sequential dependency between these use cases by simply using an additional scenario that defines a grammar (i.e., a set of valid sequences) of events that sequentially order the use cases to which these events are associated. There are two points to be made about this observation.

First, whereas TOTEM addresses sequential ordering between use cases, our approach is more fine-grained in that it also enables sequential ordering between specific paths of use cases. Semantically this is more precise and more desirable inasmuch as it allows us to eliminate some path combinations across use cases to which the sequential ordering does not apply. For example, we insist, there should be no sequential ordering between attempting to create a user who already exists and any paths associated with user deletion. But TOTEM dealing with use cases as atomic concepts cannot prevent such incorrect ordering (which may lead to test cases of dubious usefulness).

Second, TOTEM, like many others, is essentially a descriptive approach to sequential orderings between use cases. Because it does not have executability, it must rely on interleaving semantics to deal with sequences of parameterized use cases whose order is non-deterministic; an approach that leads to an explosion in the number of test cases. In contrast, in ACL, the use of parameterized events allows the generation of test paths to be more precise:

- User addition involves two paths but only the creation of a new user is relevant to the sequential ordering. Test cases for both paths of the scenario are generated

through the path sensitization of the two branches of the ***choice*** statement of that scenario.

- Similarly, the generation algorithm will create specific tests for the different branches of scenario ***removeUser***.
- *Independently* of these generated test cases, the generation algorithm will need to exercise scenario ***userDeletionFollowsUserCreation***. The only valid path through this scenario is through the creation and deletion of a *same* user. And the only invalid path through this scenario consists in having a user created but the *same user* never deleted (i.e., the ***terminate*** of the scenario never succeeds). That is, the event ***userDeletionCompleted*** is never fired with the id of a properly created user. In this case, the scenario will fail and a scenario violation will be reported in ACL, as it indeed should.
- As discussed earlier, several branches of scenario ***removeUser*** do not fire ***userDeletionCompleted***. But, given the sequential dependency can only be triggered by a properly created user, it does not have to consider the deletion of a user that does not exist. That is, given a valid user u1, the invalid paths for the sequential dependency are i) u1 does not have borrowing privileges (i.e., must pay a fine) and ii) u1 has outstanding loans.

Ultimately, the point to be grasped is that several sequences of user creations and deletions that *will* be generated (incorrectly, in our opinion) by TOTEM (e.g., create user u1 then delete user u2, create user u1 then attempt to delete user u2 who has outstanding loans, etc.) will not in our approach. We will revisit this point when

recapitulating at the end of this chapter. For now, we continue the case study following the order of presentation of TOTEM. So, the next set of use cases pertain to the creation and deletion of titles. Our model for title creation follows:

```
/*~~~Create  a title~~~*/
String newTitleInfo = null;
// stores title information such as isbn, title, authors, etc.
Observability Boolean IsTitleInfoComplete(String titleInfo);
Observability Boolean DoesTitleExist(String titleInfo);
```

```
Responsibility String  initiateNewTitleAddition()
{
        Pre(newTitleInfo ==  null) ;

        // gather system state BEFORE title creation
        statesOfUsers = getUsersStates();
        statesOfTitles = getTitlesStates();
        statesOfItems = getItemsStates();
        statesOfLoans = getLoansStates();
        Execute();
        newTitleInfo = value;
        // set to the serialized version of what was input from terminal
        Belief titleCreationStarted("Librarian has started the title creation through
                                        the Library Terminal");
        Belief librarianEntersNewTileInfo("Librarian was prompted to enter the title
                                                info through the Library Terminal");
        Post(newTitleInfo not= null);
        fire(titleCreationRequested(newTitleInfo));
}

Responsibility Integer createTitle(String titleInfo)
{
        Pre(titleInfo not= null);
        Execute();
        Post(DoesTitleExist(titleInfo) == true);
        Post(value >0);
        //the method bound to this responsibility is to return the isbn of the title
        fire(newTitleCreated(titleInfo));
}
Scenario addTitle
{
        String aTitleInfo;
        Trigger(observe(titleCreationRequested(aTitleInfo)));
        choice (IsTitleInfoComplete(aTitleInfo)) true
        {
                choice (DoesTitleExist(aTitleInfo)) false
                {
                        observe((newTitleCreated(aTitleInfo));
                        // check title creation has not modified incorrectly previous state of
                        //the system
                        statesOfTitles.AddPair(aTitleInfo, getTitleState(aTitleInfo));
                        choice (  (statesOfUsers != getUsersStates())  |
                                  (statesOfTitles != getTitlesStates())    |
                                  (statesOfItems != getItemsStates())        )  true
                        {
                        Error("UI should inform the librarian that the System has
                        incorrectly modified objects that should have not been modified.");
                        }
```

```
                    fire(titleAdditionCompleted(aTitleInfo))
                }
                alternative (true)
                {
                    Belief informTitleAlreadyExists("Librarian should be informed
                    that there already exists a title with the provided title-info");
                }
        }
        alternative (false)
        {
        Belief informTitleInfoIncomplete("Librarian should be informed through that
                                    title information is incomplete ");
        }
        newTitleInfo = null;
        Terminate();
}
/*~~~End Create  a title~~~*/
```

This portion of the model is quite similar to user creation. Observability ***IsTitleInfoComplete()*** verifies that the given ***titleInfo*** has all the attributes required to create a title. Observability ***DoesTitleExist()*** returns ***true*** if title exists for the given titleInfo; otherwise, it returns ***false***. Two responsibilities follow. Responsibility ***initiateNewTitleAddition()*** is bound to an IUT method that accepts the information required to create a new title. This information is entered by librarian via the library terminal. The precondition ***(newTitleInfo == null)*** ensures that the last title deletion request was processed before the current title creation request is processed. This variable is set to null at the end of the scenario for creation. Relevant state information is then gathered. The postcondition ensures that some title information (though not necessarily complete) has been entered. The scenario ***addTitle*** is equivalent to the use case "Add Title" in Appendix B of TOTEM. A new scenario instance is created every time an event ***titleCreationRequested*** is

fired. The different branches of this scenario are self-explanatory given the ***Belief*** and Error statements. The generation algorithm must generate a test case for each branch.

Next, title deletion:

```
/*~~~Delete  a title~~~*/

String titleInfoForDeleteTitle = null;
Observability Integer FindTitle(String titleInfo);
Observability Boolean TitleHasLoans(Integer isbn);
Observability Integer GetNumberOfItems(String titleInfo);
Observability Boolean DoesItemExist(Integer itemId);
Observability List Integer GetItems(Integer isbn);

Responsibility String initiateTitleDeletion(String titleInfo)
{
        Pre(titleInfoForDeleteTitle == null);
        Pre(titleInfo not=null);
        // gather system state BEFORE title deletion
         statesOfUsers = getUsersStates();
         statesOfTitles = getTitlesStates();
        statesOfLoans = getLoansStates();

        // since title deletion MAY entail item deletion, we do not verify
        // the immutability of the items and their states
        Execute();
        titleInfoForDeleteTitle = value;
        Post(titleInfoForDeleteTitle not= null);
        fire(titleDeletionRequested(titleInfoForDeleteTitle));
}

Responsibility removeTitle(String titleInfo)
{
        Pre(titleInfo not=null);
        Execute();
        Post(DoesTitleExist(titleInfo) == false);
        fire(titleDeleted(titleInfo));
}

Responsibility   deleteItems(Integer isbn, Integer copyid) ;
{
        Pre(isbn not= null);
        Pre(itemId>0);
        Execute();
        Post(DoesItemExist(itemId) == false);
        fire(itemDeletioncompleted(itemId);
 }
```

```
Scenario removeTitle
{
        String newInfoForDeletionTitle;
        Trigger(observe(titleDeletionRequested(newInfoForDeletionTitle)));
        choice (DoesTitleExist(newInfoForDeletionTitle)) true
         {
                 Belief displayTitleInfo("Title information should be displayed on
                terminal");
                isbn = FindTitle(newInfoForDeletionTitle);
                 choice (TitleHasLoans(isbn)) false
                {
                        each (GetItems(isbn))  { deleteItems(isbn, iterator)
                }
         observe(titleDeleted(isbn));

         // check title deletion has not modified incorrectly users or titles
        statesOfTitles.RemovePair(newInfoForDeletionTitle,
        getTitleState(newInfoForDeletionTitle));
        choice (  (statesOfUsers != getUsersStates())  |
                   (statesOfTitles != getTitlesStates())    |
                   (statesOfLoans != getLoanStates())          )  true
        {
                 Error("UI should inform the librarian that the
                 System has incorrectly modified objects that should have not been
                modified.");}
                fire(titleDeletionCompleted(newInfoForDeletionTitle))
         }
         alternative (true)
         {
                Belief informTitleHasLoanedItems("Librarian should be informed that
                title cannot be deleted because some user has loan against the title");
         }
         }
         alternative (false)
         {
                Belief informTitleDoesntExist("Librarian should be informed that title
                                                        doesn't exist");
         }
         titleInfoForDeleteTitle = null;
         Terminate();
}

Scenario removeTitleFollowsAddTitle
{
        String newTitleInfo;
        Trigger(observe(titleAdditionCompleted(newTitleInfo)));
        Terminate(observe(titleDeletionCompleted (newTitleInfo)));
}
```

/*~~~End Delete a title~~~*/

Title deletion is slightly more complex as, following TOTEM, some queries require the isbn of a book rather than its complete title information. Three observabilities are defined. Observability ***FindTitle()*** returns the ***isbn*** unique to that title, if a title exists for the given ***titleInfo***, otherwise it returns ***false***. Observability ***TitleHasLoans()*** returns ***true*** if any of the title's items (which, recall, is the term used in TOTEM for the copies of a title) has been loaned out, ***false*** otherwise. Observability of ***GetNumberOfItems()*** returns the total number of items associated with a title.

Following the observabilities three responsibilities are defined. Responsibility ***initiateTitleDeletion()*** is bound to a method that handles the incoming title deletion request. The information **titleInfo** for the title to be deleted is provided through library terminal. The method captures the input and returns it in serialized string format. Within ***initiateTitleDeletion()*** there are two preconditions. The first, namely ***(titleInfoForDeleteTitle == null)***, ensures that before the current title deletion request is processed, the last title deletion request was processed completely. The variable ***titleInfoForDeleteTitle*** is made to be null, later in scenario ***removeTitle***, just before the scenario terminates. The second precondition ***(titleInfo not=null)*** is used to make sure that parameter ***titleInfo*** is valid. Next, incorrect state changes are tested. After the ***Execute*** statement, the complete information for the title is returned by the bound method and stored in ***titleInfoForDeleteTitle***. The post condition verifies this information is not null. Then event ***titleDeletionRequested*** is fired.

The second responsibility is ***removeTitle()*** is bound to a method that is responsible for actually deleting the title for the given titleInfo. It is self-explanatory.

The third responsibility ***deleteItems()*** is bound to a method that is responsible for actually deleting the given item for a title with given isbn. Another responsibility removeItem()(discussed later) is similar to this responsibility, distinction between these two can be made based upon the manner these responsibility would execute. The responsibility ***deleteItems()*** executes when IUT deletes the copies, without any manual intervention from librarian(this is done as part of title deletion task, to make sure that all the copies of a title have been deleted, before the title itself can be deleted) whereas the responsibility removeItem() executes when copies are deleted because librarian has manually selected a copy to be deleted. One point to be underscored here is that, execution of responsibility deleteItems() doesn't trigger the scenario removeItem.

The pre and post conditions of responsibility ***deleteItems()*** are for standard sanity check of input and the result and are self-explanatory. Event itemDeletionCompleted(itemId) is fired from this responsibility, after the post condition verification. The same event is also fired from scenario removeItem, to signify the successful item deletion and it is observed by scenario itemLifeTime to mark the end of item's existence. Since execution of responsibility deleteItems() doesn't trigger the scenario removeItem, it can't be relied upon to supply the itemDeletionCompleted event to itemLifeTime scenario, in case of automated deletion of copies by IUT hence this event should be fired from responsibility deleteItems() as well.

The s***cenario removeTitle*** is equivalent to the use case “Remove Title” in Appendix B of TOTEM (without the test for reserved items, which we do not model. Should we have, it would merely two more branches to the scenario.) The complexity of this use case comes from the fact that in TOTEM, if a title has no outstanding (reservations or) loans, then deleting it entails deleting all copies of it, as will be tackled shortly. The trigger of this

scenario ensures the variable ***newInfoForDeletionTitle*** gets set for each scenario instance, that is, each time event ***titleDeletionRequested*** is fired. Given the ***Belief*** statements throughout this scenario, its branches are self-explanatory. The one thing to notice is that, following TOTEM, queries on the title are made via its isbn, which is obtained using observability ***FindTitle***. As usual, the algorithm for generating test cases will produce one test case per branch. The branch of interest is the one corresponding to a valid deletion (i.e., after checking there are no outstanding reservations or loans). It uses the observability ***GetItems()*** to get from the IUT the list of ids of items (i.e., copies) associated with the isbn to delete. Then, for this branch to succeed, the deletion of each copy of the title to delete must be observed by ACL/VF. However, contrary to individual deletions initiated by the librarian via the terminal, these deletions are to be carried out by the system automatically as part of the functionality for deleting a title. Ideally, semantically, the order in which these copies are deleted should be irrelevant. However ACL does not currently support syntax that allows the system to observe a set of invocations of responsibilities in any order (like co-regions in Message Sequence Charts (online)). Consequently our model imposes a particular order of deletion for copies: they must be deleted by the IUT in the order captured in the return value of ***GetItems()***. To do so, we use ACL's ***each*** statement, which traverses the set of ids returned by ***GetItems()***, accessing each one via the keyword ***iterator***. (We are *not* assuming that a title has a set of continuous ids for its copies, a detailed not addressed in TOTEM.) This ***each*** statement requires that a call to the method bound to responsibility ***deleteItems()*** be performed for each copy. Next, the key-value pair for the title being deleted is removed from the variable ***statesOfTitles and incorrect state changes are then checked***. Once all the required deletions have been observed, this branch of the scenario fires event

titleDeletionCompleted, which is used in the next scenario. This scenario, namely ***removeTitleFollowsAddTitle***, addresses the sequential relationship between adding and removing a title in the same simple way as for user creation and deletion.

At this point of the discussion, two remarks are in order. First, clearly, an ACL specification being executable and testable, it is at a level of abstraction lower than a use case. To illustrate this, consider the description in TOTEM of the use case for deleting an item:

> **Use case name**: Remove Title
> **Participating actor**: Librarian
> **Parameters**: in Isbn: Integer
> **Entry condition**:
> 1. The librarian requests the title information from the librarian terminal. The Find Title use case is used. If the title doesn't exist, then proceeds to 3.
> **Flow of events**:
> 2. The title information is displayed on the librarian terminal. If the title is reserved or some of the title's loan copies are loaned, the librarian is not allowed to remove the
> title.
> **Exit condition**:
> 3. The title has been removed, along with all the items associated with the title or the librarian is not allowed to remove the title.

This use case is very representative of all use cases of TOTEM paper. It suffers from several small semantic glitches: a) it is not specified how the parameter is used within the use case, b) "Find Title" is not in the set of supplied use cases nor in Figure 2 of TOTEM. In fact, it is merely a step of this use case and, as such, does not belong in a pre-condition, c) semantically, to 'proceed' from a pre- to a post-condition is incorrect: a use case has to have some path through it. In other words, the wording here is somewhat infelicitous, d) the flow of events conveys the possible paths through the use case in a very (nay too) succinct way.

The point to be grasped is that generating tests from such a description is *not* automatable. TOTEM remedies this by assuming that there is a sequence diagram for each

such use case. But the diagrams being used (e.g., Figure 6 of TOTEM) do not use UML 2.0 syntax and do not capture all possible paths through a use case. Ultimately, this confirms that TOTEM does not offer a tool that can generate a test plan consisting of paths to cover across a set of use cases.

Second, in contrast with TOTEM, an ACL specification can be used to generate such a test plan. But there is a price: testability (via executability) requires precise semantics. That is, what is to be observed during the execution of an IUT must be precisely defined. This 'forces' the modeler to address details that are typically not viewed as relevant to system-level testing. For example, forcing the IUT to delete the copies of a title in a particular order is a detail that should have been irrelevant at this level of abstraction. The solution to this glitch is, as previously mentioned, the implementation of new ACL syntax that could accommodate observing a set of responsibilities (or more precisely the corresponding invocations of the methods bound to these responsibilities) in any order. Current experimentation with a JavaMOP (online) implementation of ACL/VF has revealed that, as usual, what in theory is feasible can get quite tricky (if not downright 'ugly') in practice. (More precisely, it appears JavaMOP would require that 'any order' be implemented as 'all possible orderings', that is, that *all* possible ordering be spelled out in the specification, which is clearly an unacceptable solution.)

Let us now consider the use cases pertaining to items per se. For item addition, TOTEM has the following definition, which is interesting because one use case refers to another. Here is the portion of our model that pertains to item addition:

```
/*~~~Add an item to a title~~~*/

String titleInfoForAddItem = null;
String titleInfoForDeleteItem = null;

Responsibility String  initiateNewItemAddition()
{
        Pre(titleInfoForAddItem == null);
        // gather system state BEFORE item creation
        statesOfUsers = getUsersStates();
        statesOfTitles = getTitlesStates();
        statesOfItems = getItemsStates();
        statesOfLoans = getLoansStates();

        Execute();
        titleInfoForAddItem = value;
        Belief itemCreationStarted("Librarian has started the item addition through
                                    the Library Terminal");
        Belief librarianEntersTitleInfo("Librarian was prompted to enter the title-info
                                info through the Library Terminal");
        Post(titleInfoForAddItem not= null);
        fire(itemAdditionRequested(titleInfoForAddItem));
}
Responsibility Integer addItem(Integer isbn)
{
        Pre(isbn > 0);
        Execute();
        Post(value > 0);  //a valid unique id must have been generated for this copy
        fire(newItemAdded(isbn, value));
}
Scenario itemAddition
{
        Integer itemId;
        Integer isbn;
        String newTitleInfoForAddItem;
        Trigger(observe(itemAdditionRequested(newTitleInfoForAddItem)));
        choice (DoesTitleExist(newTitleInfoForAddItem)) true
        {
          Belief displayTitleInfo("Title info should be displayed on terminal");
          Belief displayItemInfo("Existing items for this title should be listed on
                                the library terminal");
        }
        alternative (false)
        {
           Belief reportTitleDoesntExist("Librarian should be informed that title
                                doesn't exist and needs to be created first");
          observe(titleAdditionComplete(newTitleInfoForAddItem));
        }
        isbn = FindTitle(newTitleInfoForAddItem);
```

```
itemId = addItem(isbn);
// check item creation has not modified incorrectly previous state of the system
statesOfItems.AddPair(itemId, getItemState(itemId));
// the addition of an item MAY change its title so our check must NOT
// require that the relevant title state remain unchanged
Dictionary(String, Integer)    temp;
statesOfTitles.RemovePair(newTitleInfoForAddItem,
getTitleState(newTitleInfoForAddItem));
temp = getTitlesStates();
temp. RemovePair(newTitleInfoForAddItem,
getTitleState(newTitleInfoForAddItem));
choice (  (statesOfUsers != getUsersStates())  |
          (statesOfTitles != temp)    |
          (statesOfItems != getItemsStates())    |
          (stateOfLoans != getLoansStates())     )  true
{
   Error("UI should inform the librarian that the
        System has incorrectly modified objects that should have not been
        modified");
}
observe(newItemAdded(newTitleInfoForAddItem,itemId);
Terminate(fire(itemAdditionCompleted(itemId)));
}

/*~~~End -Add an item to a title~~~*/
```

As we progress through the case study, in order to avoid long somewhat redundant explanations, we will assume that, because most of the model is similar to previously explained material, it is sufficiently self-explanatory. We will thus focus only on specific details. Here, the one thing to notice is that, as per the TOTEM use case, should a title not exist before a copy is added to it, the use case for adding this title needs to be observed. Whereas this may be quite complex to enforce in other approaches, it is trivial in this model due to the fact that our modeling strategy uses the firing of an event to signal the successful completion of a use case (this event being used to enforce any sequential ordering involving this use case). Another point to notice within this scenario is that an item addition may actually change its own title's state, which has been addressed in our model.

Let us now give the model for item deletion.

```
/*~~~Delete an item from a title~~~*/

Observability Boolean IsItemAvailable(Integer itemId);

Responsibility String  requestItemDeletion()
{
        Pre(titleInfoForDeleteItem == null);
        Pre(itemIdToBeDeleted == 0);
        Execute();
        titleInfoForDeleteItem = value;

        // gather system state BEFORE item deletion
        statesOfUsers = getUsersStates();
        statesOfTitles = getTitlesStates();
        statesOfItems = getItemsStates();
        statesOfLoans = getLoansStates();

        Post(titleInfoForDeleteItem not= null);
        fire(itemDeletionRequestedForATitle(titleInfoForDeleteItem));
}
Responsibility Integer returnSelectedItemIdForDeletion(String titleInfo)
{
        Pre(titleInfo not= null);
        Execute();
        Post(value >0);
        itemIdToBeDeleted = value;
        fire(itemDeletionRequested(itemIdToBeDeleted));
}
Responsibility Integer removeItem(Integer itemId)
{
        Pre(itemIdToBeDeleted > 0);
        Pre(itemId > 0);
        Execute();
        Post(DoesItemExist(itemId) == false);
        fire(itemDeleted(itemId));
}
Scenario removeItem
{
        String newTitleInfoForDeleteItem;
        Integer newItemIdToBeDeleted;
        Trigger(observe(itemDeletionRequestedForATitle(newTitleInfoForDeleteItem)));
        choice (DoesTitleExist(newTitleInfoForDeleteItem)) true
        {
                Belief displayTitleInfo("Title information should have been displayed on
                the library terminal ");
                Belief itemSelected("Librarian should have selected the item to be
                deleted");
```

```
                newItemIdToBeDeleted=
                        returnSelectedItemIdForDeletion(newTitleInfoForDeleteItem);
                choice (IsItemAvailable(newItemIdToBeDeleted)) true
                {
                        observe(itemDeleted(newItemIdToBeDeleted));
                        // check item deletion has not modified incorrectly previous state
                        //of the system
                        statesOfItems.RemovePair(newItemIdToBeDeleted,
                        getItemState(newItemIdToBeDeleted));
                        // the deletion of an item MAY change its title so our check must
                        //NOT require that the relevant title state remain unchanged
                        Dictionary(String, Integer)    temp;
                        statesOfTitles.RemovePair(newTitleInfoForDeleteItem,
                                        getTitleState(newTitleInfoForDeleteItem));
                        temp = getTitlesStates();
                        temp. RemovePair(newTitleInfoForDeleteItem,
                        getTitleState(newTitleInfoForDeleteItem));
                        choice (  (statesOfUsers != getUsersStates())  |
                                (statesOfTitles != temp)    |
                                 (statesOfItems != getItemsStates())     |
                                 (statesOfLoans != getLoansStates())    )  true
                        {
                                Error("UI should inform the librarian that the
                                System has incorrectly modified objects that should have
                                not been modified.");
                        }
                        fire(itemDeletionCompleted(newItemIdToBeDeleted));
                }
                alternative (false)
                 {
                        Belief informItemIsOnLoan("librarian should be informed through
                        the  terminal, that item is not available and hence cannot be
                        removed");
                 }
  }
  alternative (false)
  {
    Belief InformTitleDoesntExist("Librarian should have been informed
                                        that title with given title-info doesn't exist");
  }
  itemIdToBeDeleted = 0;
  titleInfoForDeleteItem = null;
  Terminate();
}
```

```
Scenario removeItemfollowsAddItem
{
        Integer newItemId;
        Trigger(observe(itemAdditionCompleted(itemId)));
        Terminate(observe(itemDeletionCompleted(newItemId)));
}
```

Observability ***DoesItemExist()*** returns ***true*** if an item exist for the given itemId, otherwise returns ***false***. Observability ***IsItemAvailable()*** returns ***true*** , if the item with given itemId has not been loaned out, otherwise returns ***false.*** The responsibility ***requestItemDeletion()*** is straightforward (including gathering state information***).*** The second responsibility ***returnSelectedItemIdForDeletion()*** is bound to an IUT method that returns the itemId for that particular item that is selected for deletion (via the terminal) by the librarian. It is because selection of the item to delete is carried out by the librarian that explains why we also need the ***deleteItem()*** responsibility that was discussed earlier (for dealing with deletions that do not involve the librarian). The third responsibility, ***removeItem(),*** is bound to an IUT method, which actually removes the item and returns the corresponding itemid.

The scenario ***removeItem*** corresponds to the use case "Remove Item" on Appendix B of TOTEM. It is triggered by observing the event ***itemDeletionRequestedForATitle,*** which is fired in responsibility ***requestItemDeletion().*** The different branches of this scenario are self-explanatory.

Next, scenario ***removeItemfollowsAddItem*** addresses the sequential relationship between adding and removing an item. It is straightforward because it merely checks that the *successful* addition of an item is eventually followed by the deletion of that item, *regardless*

of the (possibly absence of) borrowing history of this item. This corresponds to the semantics put forth by TOTEM.

The last aspect of interest of the TOTEM case study is the handling of loans. As can be seen in Figure 2 of TOTEM, this involves several use cases and several dependencies. We start with the borrowing use case:

```
/*~~~Borrow Item~~~*/

Integer loanRequestedForUserId=0;
Observability Integer GetUserLoanCount(Integer userId);
Observability String GetTitleInfoForItem(Integer itemId);
Observability Boolean IsItemAReferenceCopy(Integer itemId);

Responsibility initiateNewLoanCreation(Integer userId, Integer itemId)
{
        Pre(loanRequestedForUserId == 0);
        Pre(loanRequestedForItemId == 0);
        Execute();
        loanRequestedForUserId = userId;
        loanRequestedForItemId = itemId;
        Post(loanRequestedForUserId>0);
        Post(loanRequestedForItemId>0);
        fire(loanRequested(loanRequestedForUserId,loanRequestedForItemId));
}
Responsibility Integer addLoan(String userId, String itemId)
{
        once Scalar Integer userLoanCount;
        userLoanCount = PreSet(GetUserLoanCount(userId));
        Execute();
        Post(GetUserLoanCount(userId) == userLoanCount + 1);
        Post(value >0);
        fire(loanAdded(userId, itemId));
}

Responsibility revokePrivilege(Integer userId)
{
        Pre(userId not=0);
        Execute();
        Post(UserHasPrivilege(userId) == false);
        fire(privilegeRevoked(userId));
}
```

```
Scenario borrowLoanCopy
{
        String titleInfo;
        Integer userLoanCount;
        Integer newLoanRequestedForUserId;
        Integer newLoanRequestedForItemId;
        Trigger(observe(loanRequested(newLoanRequestedForUserId,
                newLoanRequestedForItemId));
choice (FindUserInfo(newLoanRequestedForUserId) not= null) true  //1
{
         userLoanCount = GetUserLoanCount(newLoanRequestedForUserId);
        titleInfo = GetTitleInfoForItem(newLoanRequestedForItemId);
        choice (userLoanCount < Parameters.MaxLoanCount) true //2
  {
        choice (FindTitle(titleInfo) > 0) true //3
        {
          choice (IsItemAvailable(newLoanRequestedForItemId)) true  //4
          {
           choice (IsItemAReferenceCopy(newLoanRequestedForItemId)) false  //5
           {
                choice (UserHasPrivilege(newLoanRequestedForUserId)) false //6
                {
                  Belief reportPrivilegeRevoked("librarian should be
                  informed that user doesn't have privilege");
                  observe(finePaid(newLoanRequestedForUserId));
                } //end 6
                observe(loanAdded(newLoanRequestedForUserId,
                newLoanRequestedForItemId));
                fire(itemBorrowed(newLoanRequestedForItemId));
                choice ( (userLoanCount+1) == Parameters.MaxLoanCount ) true
                {
                  observe(privilegeRevoked(newLoanRequestedForUserId));
                }
            }
             alternative (true) //5
             {
                Belief reportItemIsAReferenceCopy("librarian should be informed
                 that item requested for loan is a reference copy");
             }
          }
          alternative (false)   //4
          {
                Belief reportItemIsNotAvailable("librarian should be informed
                that item has been loaned to some other user");
          }
        }
        alternative (false)  //3
        {
                Belief reportTitleDoesntExist("librarian should be informed
```

```
                that title doesn't exist");
        }
  }
  alternative (false)  //2
  {
        Belief reportMaximumLoanTaken("librarian should be informed that
        user has borrowed maximum allowed number of items ");
  }
}
alternative (false)  //1
  {
        Belief reportUserDoesntExists("Librarian should be informed that user
        doesn't exist");
  }
 loanRequestedForUserId = 0;
 loanRequestedForItemId = 0;
 Terminate();
}

Scenario borrowLoanFollowsAddUserAndAddItem
{
        Integer newUserId;
        Integer newItemId;
        Trigger(observe(loanRequested(newUserId, newItemId));
        choice (DoesUserExist(newUserId)) true
        {
                choice (DoesItemExist(newItemId))true
                {
                        observe(loanAdded(newUserId, newItemId));
                }
        }
        Terminate();
}
/*~~~End Borrow Item~~~*/
```

Observability ***GetUserLoanCount()*** is used to learn how many loans the user holds. The second observability ***GetTitleInfoForItem()*** is used to get the title information for which the item will be borrowed. Observability ***IsItemAReferenceCopy()*** returns ***true*** if the item is a reference copy, then ***false*** otherwise. For responsibility ***initiateNewLoanCreation()*** a user's ***userid*** and the loan item's ***itemId*** are provided through the library terminal and stored in contract variables. Event ***loanRequested*** is fired. The second responsibility, ***addLoan()***, is

bound to a method that will create the loan and return the loanId. The ***once*** modifier is used to ensure that the specified variable is assigned a value only one time. The ***PreSet*** statement is used to set a value for this variable before the responsibility is executed. This value can then be used within the body of the responsibility or within a post statement. The observeability ***GetUserLoanCount()*** fetches the current loan count for the given user and makes sure that the user's current loan count has been increased. In postcondition ***(value >0),*** it is assumed that every ***loanId*** returned is unique and is a positive integer. An event ***loanAdded*** is fired to signify that the loan has been added for the given userId and itemId. The third responsibility, ***revokePrivilege***(), is bound to the method of IUT and revokes the privilege of the user with the given userId.

The scenario ***borrowLoanCopy*** corresponds to the use case ***borrowLoanCopy*** of TOTEM. The scenario is triggered every item there is a borrow request. In the first choice statement, the observability ***FindUserInfo()*** that the requesting user is valid. Then, scenario variable ***userLoanCount*** is set to the current loan count for the given user. Observability ***GetTitleInfoForItem()*** then evaluates and gets the title information for which the borrowed loan is requested and stores it in variable ***titleInfo***. The next choice ensures that the ***userLoanCount*** is less than the maximum allowed number of loans. The request loan will not be allowed if a user already has the maximum number of loans. In next choice, ***FindTitle()*** checks if the title exists. Then ***IsItemAvailable()*** verifies the item to be borrowed is not be already on loan. We then check if the item is a reference. All invalid paths have their own branches that are self-explanatory. This cascade of choice points avoids the complex generation algorithm that would be required in order to deal with a complex combinatorial statement capturing all these conditions (see Binder, 2000). At this point, we must check if the

user has borrowing privileges. TOTEM states that the user *may* pay the fine and that the request fails if s/he does not. Such a statement is *not* verifiable because it lacks a sequential constraint: after how much time must the system decide whether user has or has not paid the fines? The good news is that ACL does offer syntax for a verifiable timer (see ACL Specification, online). It is therefore entirely possible to implement a timing constraint in the IUT after which it is deemed the user has declined to pay. But this would merely introduce another choice point and another invalid path in the scenario and, more importantly, introduce slightly more complexity to the IUT used to run the specification for verifying our model. From a test generation viewpoint, this additional constraint is of little interest: it merely introduces one additional level of branching.

Consequently, we have instead chosen to depart slightly from the TOTEM example and model the requirement that the user *must* pay the fine in order to proceed with the loan (which remains in a 'pending' state otherwise). We acknowledge that having such a possibly everlasting pending state is not realistic but this allows us to illustrate more interesting path generation case than just yet another invalid path (and extra level of branching). Let us elaborate. The next ***choice*** statement checks if the user has no borrowing privilege. In that case, a ***Belief*** is used to notify the librarian and the scenario 'hangs' until the event ***finePaid*** is observed. Given no sequential constraint is specified, this scenario will be violated if the execution of the IUT terminates without the fine of this user being paid. Otherwise, the scenario will proceed once the fine is paid. At that point, there is no longer any issue with borrowing privilege and the scenario requires the event ***loanAdded*** to be observed. This event's purpose is to capture the loan having been recorded. It is therefore somewhat surprising to see that this ***observe*** statement is followed by firing event ***itemBorrowed***. This is

the modeling detail we want to explain. Namely, in ACL, upon being fired, an event is broadcasted to all *triggered* scenarios that observe it. Once observed by a scenario, this event cannot be observed again. This explains why this scenario fires event ***itemBorrowed***. We must add that this actually simplifies the generation of tests since it avoids repeated observation of events, which can be error-prone. Finally, this branch of the scenario uses a choice to check whether the user has reached the allowed maximum number of loans, in which case the event ***privilegeRevoked*** is fired.

The scenario ***borrowLoanFollowsAddUserAndAddItem*** must capture that a successful loan creation must be preceded by a successful user creation and a successful item creation. This scenario is triggered by each new loan requested. Through the ***observe*** statement, the scenario acquires a user id and an item id. An initial idea may be to observe event ***userAdditionCompleted*** with the correct user id in order to verify the user has been successfully created. This will not work for two reasons: a) that event could have been followed by user deletion and b) as just explained, this event cannot be observed here since this scenario was not triggered when the firing of ***userAdditionCompleted*** occurred. The conclusion of such analysis is that we instead simply need to check that, at the time of the loan request, both the user and the item exist in the system, which implies they were successfully added. This explains the rest of this scenario: having verified the validity of the user and copy, this scenario succeeds if event ***loadAdded*** can be observed (which implies the loan was successfully created as per scenario ***borrowLoanCopy***).

The next use case to tackle is the return of an item. The model for this is straightforward.

```
/*~~~Return Item~~~*/
```

```
Integer returnLoanRequestedForUserId=0;
Integer returnLoanRequestedForItemId=0;

Observability Integer FindLoan(Integer userId, Integer itemId);
Observability Boolean IsLoanOverdue(Integer loanId);
Observability Integer LoanOverduePeriod(Integer loanId);
Observability Boolean DoesUserHaveLateFee(Integer userId);

Responsibility initiateReturnLoan(Integer userId, Integer itemId)
{
        Pre(returnLoanRequestedForUserId == 0);
        Pre(returnLoanRequestedForItemId == 0);
        Execute();
        returnLoanRequestedForUserId = userId;
        returnLoanRequestedForItemId = itemId;
        Post(returnLoanRequestedForUserId>0);
        Post(returnLoanRequestedForItemId>0);
        fire(returnLoanRequested(returnLoanRequestedForUserId,
                returnLoanRequestedForItemId));
}

Responsibility Integer returnLoan(String userId, String itemId)
{
        once Scalar Integer userLoanCount;
        userLoanCount = PreSet(GetUserLoanCount(userId));
        Pre(userId > 0);
        Pre(itemId > 0);
        Execute();
        Post(GetUserLoanCount(userId) == userLoanCount - 1);
        Post(value >0);
        fire(loanDeleted(userId, itemId));
}
Responsibility recordFine(Integer loanId)
{
        Pre(loanId > 0);
        Execute();
        Post(value >0);
        Post(DoesUserHaveLateFee(value)== true);
        fire(fineRecorded(value, loanId));
}
Scenario returnLoanCopy
{
        Integer loanId;
        Integer newReturnLoanRequestedForUserId;
        Integer newReturnLoanRequestedForItemId;
        Trigger(observe(returnLoanRequested(newReturnLoanRequestedForUserId,new
                                        ReturnLoanRequestedForItemId)));
        loanId = FindLoan(newReturnLoanRequestedForUserId,
```

```
                                                    newReturnLoanRequestedForItemId);
        choice (loanId > 0)true
        {
                Belief displayLoanInfo("Loan information should be displayed on library
                terminal");
                choice (IsLoanOverdue(loanId)) true //1
                {
                        observe(fineRecorded(newReturnLoanRequestedForUserId,
                                loanId));
        choice (LoanOverduePeriod(loanId) >=
                                        Parameters.MaxLoanInterval) true
        {

        observe(privilegeRevoked(newReturnLoanRequestedForUserId)); /*User's
                        privilege should be revoked if this loan has been overdue for more
                        than allowed interval*/
        }
        }
        choice (DoesUserHaveLateFee(newReturnLoanRequestedForUserId)) true //2
        {
                observe(finePaid(newReturnLoanRequestedForUserId);
        }
        observe(loanDeleted(newReturnLoanRequestedForUserId,
                                newReturnLoanRequestedForItemId));
        fire(loanDeletionCompleted(newReturnLoanRequestedForUserId,
                                newReturnLoanRequestedForItemId));
        fire(itemReturned(newReturnLoanRequestedForItemId));

        }
        alternative (false)
        {
                Belief reportLoanDoesntExist("librarian should be informed through library
                                        terminal that there exists no loan for given item for
                                        this particular user");
        }
        returnLoanRequestedForUserId = 0;
        returnLoanRequestedForItemId = 0;
        Terminate();
}
/*~~~End -Return Item~~~*/
```

Observability ***FindLoan(itemId)*** returns loan information, which is issued against ***userId*** and ***itemId***. Observability ***IsLoanOverdue()*** returns ***true*** if a loan is overdue. Observability ***LoanOverduePeriod()*** returns how many days overdue the item is. Responsibility ***returnLoan()*** is bound to IUT method that deletes the loan and return the

loanId for the deleted loan. The ***once*** modifier is used to ensure that the specified variable is assigned a value only one time. This keyword indicates that the variable being declared can only be assigned to Once. The ***userLoanCount*** variable is used to assigned a value only once before the responsibility execution. The ***Preset*** statement causes the value stored in the variable to persist over the responsibility execution. The third responsibility ***recordFine()*** is bound to an IUT method that calculates the fine associated for the given loan.

The scenario ***returnLoanCopy*** is triggered every time a return is requested. Observability ***FindLoan()*** is then used to retrieve the corresponding loanId (which was set in ***addLoan()***). If this loanId is not valid, then an appropriate message to the librarian will be displayed. In the case of a valid loan, we then check if the loan is overdue, if so, then it cannot be removed. Instead we must observe event ***fineRecorded()***, which is fired in responsibility ***recordFine()*** that records the fine. We must also check whether the loan is so overdue that the borrowing privilege must be revoked. Next, an Observability ***DoesUserHaveLateFee()*** is then used to check, if the user has any late fee to pay which has just been recorded now or at some point of time in past, if so, then user must pay fine before the loan can be removed, this is ensured by observing the event finePaid() event inside the choice block. If user doesn't have any late fee to pay or has cleared it by paying the fine, then we must observe event ***loanDeleted()***, which is fired in responsibility ***returnLoan()***. At that point, events ***loanDeletionCompleted*** and ***itemReturned*** are fired. These events are respectively used in scenarios ***loanLifeTime*** and ***itemLifeTime*** that will be discussed later.

Use case "Return Loan Copy" has several sequential relationships in Figure 2 of TOTEM (also our Figure 2). Two of them have already been dealt with:

1) In scenario ***removeUser***, we do not use events to enforce that the user must not have outstanding loans nor a revoked borrowing privilege in order to be deleted. Instead we use choice statements to ensure this is the case. As explained above, observing an event would not work as we need to verify at a particular point in the execution.

2) Similarly, an item cannot be removed unless that item is available (i.e., not on loan). This is not enforced using events but rather with observability ***IsItemAvailable()***, which queries the IUT for a status at a specific point in time.

In order to address the remaining sequential relationships of Figure 2, we must first model the use cases Collect Fine and Renew Loan. Let us start with the latter:

```
/*~~~Renew Loan~~~*/

Integer renewLoanRequestedForUserId=0;
Integer renewLoanRequestedForItemId=0;
Observability Integer GetLoanRenewalCount(Integer userId, Integer itemId );
Observability Integer GetLoanID(Integer userId, Integer itemId);
Responsibility initiateRenewLoan(Integer userId, Integer itemId)
{
        Pre(renewLoanRequestedForUserId == 0);
        Pre(renewLoanRequestedForItemId == 0);
        Execute();
        renewLoanRequestedForUserId = userId;
        renewLoanRequestedForItemId = itemId;
        Post(renewLoanRequestedForUserId>0);
        Post(renewLoanRequestedForItemId>0);
        fire(renewLoanRequested(renewLoanRequestedForUserId,
                renewLoanRequestedForItemId));
}
```

```
Responsibility Integer renewLoan(Integer userId, Integer itemId)
{
        once Scalar Integer loanRenewalCount;
        loanRenewalCount = PreSet(GetLoanRenewalCount(userId, itemId));
        Pre(userId > 0);
        Pre(itemId > 0);
        Execute();
        Post(GetLoanRenewalCount(userId, itemId) == loanRenewalCount + 1);
        Post(value >0);
        fire(loanRenewed(userId, itemId));
}
Responsibility reportExpiredLoan(Integer userId, Integer itemId)
{
        Pre(userId > 0);
        Pre(itemId > 0);
        Execute();
        fire(loanExpired(userId, itemId));
}
```

```
Scenario renewLoan
{
        Integer loanId;
        Integer loanRenewalCount;
        Integer newRenewLoanRequestedForUserId;
        Integer newRenewLoanRequestedForItemId;
        Trigger(observe(renewLoanRequested
        (newRenewLoanRequestedForUserId,
          newRenewLoanRequestedForItemId)));
        loanId=FindLoan(newRenewLoanRequestedForUserId,
                        newRenewLoanRequestedForItemId);
        choice (loanId > 0) true
         {
                Belief displayLoanInfo("Loan information should be displayed on
                the library terminal");
                loanRenewalCount=
                GetLoanRenewalCount
                (newRenewLoanRequestedForUserId,
                newRenewLoanRequestedForItemId);
                choice (UserHasPrivilege(newRenewLoanRequestedForUserId))
                        false
                {
                        observe(finePaid(newRenewLoanRequestedForUserId));
                }
                choice (loanRenewalCount <
                                        Parameters.MaxLoanRenewalCount) true
                 {
                        observe(loanRenewed(
                        newRenewLoanRequestedForUserId,
                        newRenewLoanRequestedForItemId));
                        fire(loanRenewedCompleted
                        (newRenewLoanRequestedForUserId,
                        newRenewLoanRequestedForItemId));
                }
                alternative (false)
                {
                        Belief reportLoanRenewedMaxTimes("Librarian should be
                        informed that loan has already been renewed maximum
                        number of allowed renewals");
                 }
         }  //of valid loan id
        alternative (false)
        {
                Belief reportLoanDoesntExist("Librarian should be informed that
                loan doesn't exist for the given userId and ItemId");
        }
        Terminate();
}
```

```
/*~~~End- Renew Loan~~~*/
```

This portion of the model should be self-explanatory as it is very similar to previous parts of the model. The only detail to point out is that, in scenario ***renewLoan***, the user *must* pay any outstanding fine before the loan is renewed.

The scenario ***renewLoanFollowsBorrowLoan*** captures that specific sequential relationship:

```
Scenario renewLoanFollowsBorrowLoan
{
        Integer RenewloanId;
        Integer loanItemId;
        Integer loanUserId;
        Trigger(observe(renewLoanRequested(loanUserId, loanItemId));
        RenewloanId = FindLoan(loanUserId, loanItemId);
        choice (RenewloanId > 0)) true
        {
                choice (IsLoanOverdue(RenewloanId)) true
                {
                        observe(finePaid(RenewloanId));
                }
                observe(loanRenewedCompleted(loanUserId, loanItemId));
        }
        Terminate();
}
```

This scenario is triggered every time a renewal request occurs. We then determine whether this is a valid loan request by finding its id. If this id is not valid, then this is not a valid renewal request and the scenario terminates (as there is no sequential relationship to enforce in this case). Conversely, for a valid renewal request, we must check if a fine needs to be paid (in the case of an overdue loan). If this is the case, then we must observe the payment of this fine (via event ***finePaid***). Whether a fine was due or not, given this is a valid renewal request we must then observe event ***loanRenewedCompleted***.

Next, we must address paying fines. Recall that, as in TOTEM, fees of a user are paid in full, not per item.

```
/*~~~Pay Fine~~~*/

Integer collectFineRequestedForUserId=0;

Responsibility initiateCollectFine(Integer userId)
{
        Pre(collectFineRequestedForUserId == 0);
        Execute();
        collectFineRequestedForUserId = userId;
        Post(collectFineRequestedForUserId>0);
        fire(collectFineRequested(collectFineRequestedForUserId));
}

Responsibility Integer payFine(String userId)
{
        Pre(userId >0);
        Execute();
        Post(DoesUserHaveLateFee(userId) == false);
        Post(value >0);
        fire(lateFeeDeleted(userId));
}

Scenario payFine
{
  Integer newCollectFineRequestedForUserId;
  Trigger(observe(collectFineRequested(newCollectFineRequestedForUserId)));
  choice (FindUserInfo(newCollectFineRequestedForUserId) not= null) true
  {
        observe(lateFeeDeleted(newCollectFineRequestedForUserId));
        fire(finePaid(newCollectFineRequestedForUserId));
  }
  alternative (false)
  {
        Belief reportUserDoesntExist ("librarian should be informed that
        user doesn't exist");
  }
  collectFineRequestedForUserId = 0;
  Terminate();
}
```

By itself, this portion of the model is self-explanatory. Two additional scenarios must be introduced to complete our model. First:

```
Scenario itemLifeTime
{
  Integer aItem;
  Trigger(observe(itemAdditionCompleted(aItem)));
  choice (DoesItemExist(aItem)) true
  {
        choice (IsItemAvailable(aItem)) true
        {
              observe(itemBorrowed(aItem)) |
                    observe(itemDeletionCompleted(aItem));
              choice (DoesItemExist(aItem)) true
              {
                    choice (IsItemAvailable(aItem)) false
                    {
                          observe(itemReturned(aItem));
                    }
              }
        }
        redo;
  }
  Terminate();
}
```

The purpose of this scenario is to capture explicitly what can happen to an item after its creation. The scenario is triggered each time a new item is created *successfully* (via the observation of event ***itemAdditionCompleted***). The scenario consists of a ***redo*** loop controlled by ***DoesItemExist()***. (That is, we loop on the condition of the ***choice*** in which the redo is contained.) An item does exist upon its creation. Upon looping, if it no longer exists, then the item has been deleted and the scenario completes successfully. Within the loop, we check if the item is available (which will be the case initially). If it is, then it will either be borrowed or deleted (as captured in the OR (|) statement). Once one of these two events occurs, the next ***choice*** checks if the item still exists. If it does not, the ***redo*** will fail and the scenario will terminate successfully: the item has been removed. If the item still exists, then it has been borrowed and the scenario will wait to observe event ***itemReturned***. After this observation the item should be now available and the redo will loop successfully. What is interesting with this

scenario is that it captures something omitted in TOTEM, namely that the pair of use cases Borrow and Return an Item can be repeated over and over for a same item. (In fact, scrutinizing Figure 4 of TOTEM, one notices that the edge DF has been mistakenly forgotten, though it does appear in Figures 2 and 5). This correction of this omission would lead to even more test paths for an approach that already generates too many.

The scenario that completes our model follows:

```
Scenario loanLifeTime
{
        Integer loanUId;
        Integer loanItemId;
        Integer loanLoanId;
        Trigger(observe(loanAdded(loanUId, loanItemId));
        loanLoanId = FindLoan(loanUId, loanItemId);
        choice (loanLoanId >0) true
        {
                observe(loanExpired(loanUId, loanItemId)) |
                observe(loanRenewedCompleted(loanUId, loanItemId)) |
                observe(loanDeletionCompleted(loanUId, loanItemId))
                loanLoanId = FindLoan(loanUId, loanItemId);
                choice (loanLoanId >0) true
                {
                        choice (IsLoanOverdue(loanLoanId)) true
                        {
                                observe(finePaid(loanUId));
                        }
                }
                redo;
        }
        Terminate();
}
```

This scenario captures how collecting fees, renewals and returns of items interact. It is much in the same flavor as the previous one. The scenario is triggered each time a loan is *successfully* created (event ***loanAdded***). The id of this loan is retrieved (via the ***FindLoan()*** observability). If this id is 0 then the loan has been deleted. This is the condition on which the ***redo*** loops. A loan can expire or be renewed or be deleted, each case being checked by its own ***observe*** statement (which are ORed together). After one of these events has occurred, we check whether the loan still exists. If it does not, then the loan has been successfully deleted:

the condition of the redo will fail and we will exit successfully the scenario. If the loan still exists but is overdue then we will wait to observe that the fine of this user has been paid. The fine being paid will in turn enable the loan to be renewed or deleted.

3.3 Recapitulation

Having completed our model we will now recapitulate our work at a more abstract level. More precisely, we need to consider:

1) how we propose to model sequential dependencies between use cases in ACL
2) how to generate tests for these dependencies from such model and how our proposal avoids the explosion of test cases observed in TOTEM

Let us first recapitulate how the use cases and sequential dependencies of Figure 2 are handled in our model.

Node A - Scenario addUser

Node C - Scenario removeUser

Node D - Scenario borrowLoanCopy

Node B - Scenario payFine

Node F - Scenario returnLoanCopy

Node E - Scenario renewLoan

Node G - Scenario itemAddition

Node H - Scenario removeItem

Node I - Scenario addTitle

Node J - Scenario removeTitle

Section User

Edge AC - Scenario userDeletionFollowsUserCreation

Edge FC - Scenario removeUser(it is verified that user doesn't have any loans before it can be deleted)

Section Loan

Edge AD - Scenario borrowLoanCopy(user existence is the very first thing that is verified before proceeding further).

Edge GD - Scenario borrowLoanCopy(item availability is verified before loan is added. if item doesn't exist then it won't be available for borrowing)

Edge DB, BF - Scenario returnLoanCopy (when user pays fine before returning the loan). Scenario loanLifeTime also covers it.

Edge DE - Scenario renewLoanFollowsBorrowLoan. Scenario loanLifeTime also covers it.

Edge DF - Scenario returnLoanCopy. Scenario loanLifeTime also covers it.

Edge BE, EB (which makes it a cycle) -Scenario renewLoan(when user pays fine before renewing the loan). Scenario loanLifeTime also covers it.

loop EE (loop of loan renewal) - Scenario renewLoan and Scenario loanLifeTime (loanLifeTime tells that a loan can be renewed over and over again and scenario renewLoan controls as to, how many times that renewal can happen)

Edge EF - Scenario loanLifeTime.

Section Item

Edge GH - Scenario removeItemFollowsAddItem. Scenario itemLifeTime also covers it.

Edge IG - Scenario itemAddition(title existence is the first thing that is verified before proceeding further)

Edge FH - Scenario removeItem (it is verified that item should be available before it can be removed). Scenario itemLifeTime also covers it.

Section Title

Edge IJ - Scenario removeTitleFollowsAddTitle

Edge HJ - Scenario removeTitle (if there any items in a title, they are removed first before title is removed)

In summary, all nodes and edges of Figure 2 of TOTEM have been modeled. Let us now look at how we created this model. There are two key elements to our approach to modeling the sequential dependencies of use cases in ACL.

First, ACL scenarios can be used to model use cases and their paths. That is, ACL offers a rich set of semantic components to model complex concurrent scenarios. For example, the University example found on the homepage for the ACL/VF tool (vf.davearnold.ca) has a student take several courses simultaneously, each course involving several assignments, midterms and exams overlapping in time. In the case study presented here, scrutinizing the appendix of TOTEM spelling out the use cases confirms the use cases of the library system, taken individually, are generally extremely simple. We have shown that these use cases can be individually modeled as ACL scenarios of a single contract for the

librarian. Should several contracts had been used we speculate that the scenarios corresponding to individual use cases would have been only slightly more complex due to extra syntax required for one contract to use a contract variable of another contract (see the University example). In other words, it's not the ACL contracts but their scenarios that are directly relevant to use case modeling. Even more importantly in the context of this thesis, we have shown that ACL scenarios can be used to also model the sequential dependencies between use cases. Put simply, ACL scenarios can fire events and thus a sequential dependency can be modeled as an ACL scenario triggered by and monitoring the ordering of one or more events each corresponding to the successful occurrence of other use cases.

Second, scenarios can be indirectly parameterized through the use of parameterized events with the ***fire*** and ***observe*** keywords of ACL. In particular, the *modus operandi* of the ***Trigger*** statement combined with the observation of a parameterized event allows for separate scenario instances to be triggered for each distinct actual value of this parameter. Thus, for example, there is a scenario instance for each user/title/item creation and for each user/title/item deletion in our case study. This is what allows the modeling of most sequential dependencies of the library to adopt a common strategy in ACL: a specific event $e_i(o_i)$ having a specific object o_i as parameter can be used to signal the successful completion of a use case involving this object o_i. In most cases, such a signal is sufficient to enforce the occurrence and, if necessary, the ordering of two or more use cases involving the same parameter. And, because such sequencing only considers successful completion of use cases (as in TOTEM), a multitude of permutations of use cases that are *de facto* eliminated. That is, because each use case is parameterized, the handling of invalid paths through this use case can be specified without any reference to the actual value of the parameter(s). Technically, each branch of a

scenario that handles an invalid path through a use case defines *de facto* an *equivalence partition* (Binder, 2000): there is no need to test all possible values of the parameter(s) in the case of an invalid path: one is sufficient as all actual values relevant to this path will be treated equivalently.

Two question remain: a) how are tests to be generated using our proposed model and b) does our approach to the generation of tests offer a solution that improves on TOTEM's?

3.4 Test Generation and Comparison

Before presenting how test generation works in our approach, let us revisit some of the details of TOTEM. In essence their approach is summarized into the four regular expressions given in Table 1 of their paper, namely:

```
A(uid).C(uid)
I(title).J(title)
I(title).G(title, item).H(item).J(title)
(A(uid)||I(title).G(title,item)).D(uid,item).F(uid,item).
      (C(uid)||H(item).J(title))
```

It is this last expression that illustrates best the root cause for the large number of test cases TOTEM generates. In that expression, the creation of a user (A) is interleaved with the sequence title creation (I) followed by item creation (G). Similarly, user deletion (C) is interleaved with item deletion (H) and title deletion (J). The authors explain there are 3 possible sequences for the first interleaving, and 3 others for the second. Then to illustrate their generation algorithm, they apply the four regular expressions above to a very small example: 2 users, 3 titles, 2 items per title, 1 loan per user, no renew or collect fine for loans, and no system monitoring. In order to 'cover' their example (that is, use at least once each

possible value of each parameter), they use an ‘arbitrary’ (their adjective) algorithm to come up with 8 sequences, namely:

Seq1: A(u1).C(u1)

Seq2: I(t1).J(t1)

Seq3: I(t2).G(t2, i21).H(i21).J(t2)

Seq4: I(t3).A(u2).G(t3,i32).D(u2,i32).F(u2,i32).C(u2).H(i32).J(t3)

Seq5: I(t1).G(t1, i11).H(i11).J(t1)

Seq6: I(t1).G(t1, i12).H(i12).J(t1)

Seq7: A(u1).I(t2).G(t2,i22).D(u1,i22).F(u1,i22).H(i22).C(u1).J(t2)

Seq8: I(t3).G(t3, i31).H(i31).J(t3)

Noticing the first two are redundant respectively with Seq7 and Seq 5, they then start discussing combination of these sequences. First Seq3 and Seq4 are combined to produce 495 interleavings since the number of interleavings between two sequences S1 and S2 of length m and n respectively is given by C(m+n, n) = (m+n)!/(n!m!). The crucial detail is that they then arbitrarily pick one of these 495 interleavings, namely S:

S: I(t3).A(u2).I(t2).G(t3,i32).G(t2,i21).D(u2,i32).H(i21).

F(u2,i32).C(u2).H(i32).J(t2).J(t3)

They then compute that there are 1820 interleavings between S and Seq5. Again, they pick *one* of these 1820 interleavings and combine it to Seq6, then one of the resulting interleavings, and combine it with Seq7 ultimately producing 18018 interleavings that include:

S^3: I(t3).I(t1).A(u1).G(t1,i11).G(t1,i12).A(u2).I(t2).G(t2,i22).

G(t3,i32).H(i11).D(u1,i22).G(t2,i21).F(u1,i22).D(u2,i32).

H(i12).H(i21).J(t1).H(i22).C(u1).F(u2,i32).C(u2).H(i32).J(t2)

.J(t3)

Clearly, these 18018 tests are *not* a complete set as they proceed from repeatedly taking only 1 interleaving of the previous combination and interleaving it with the next sequence.

Our claim is that considering use case independence can drastically reduce the number of test cases to generate. In the library system, use case independence allows us to consider that the 3 possible orderings for (A || I.G) are equivalent, as are the three for (C || H.J). In turn, the independence of A, I and G, and of C, H and J, which can be verified using our ACL model, allows us to avoid limit the use of interleavings to only those use cases that cannot be shown to be independent. In TOTEM's small scoped example, this means considering interleavings only between instances of use cases D (borrowing) and F (returning).

In the context of this example, our generation algorithm will:

1) Carry out all relevant user, title and item creation, namely: A(u1), A(u2), I(t1), I(t2), I(t3), G(t1, i11), G(t1, i12), G(t2, i21), G(t2, i22), G(t3,i31) and G(t3,i32). Here the only ordering that matters proceeds from enforcing valid sequential dependencies, which requires the instances of I to precede those of G.
2) Consider the possible interleavings of the only two loans this scoped example deals with, namely the interleavings of (D(u1,i22).F(u1,i22)) with (D(u2,i32).F(u2,i32)).
3) Finally, carry out all relevant user, title and item deletion. Here, since deleting a title automatically deletes its copies, in principle order does not matter. However, to avoid unsuccessful instances of use cases (that would result from attempting to delete a copy of

a title that has already been deleted), our generation algorithm obeys the sequential dependency between J and H and correctly deletes items and then titles.

For the two loans of this scoped example, there are 6 interleavings:

D(u1,i22).F(u1,i22). D(u2,i32).F(u2,i32)

D(u1,i22). D(u2,i32). F(u1,i22).F(u2,i32)

D(u1,i22). D(u2,i32). F(u2,i32). F(u1,i22)

D(u2,i32).F(u2,i32). D(u1,i22).F(u1,i22)

D(u2,i32). D(u1,i22). F(u2,i32). F(u1,i22)

D(u2,i32). D(u1,i22). F(u1,i22). F(u2,i32).

The generation of these interleavings (which respect the sequential dependency that any D(i,j) must precede the corresponding F(I,j)) is fully automated and determines the number of tests to run for this very limited example. In other words, there are 6 test cases to run that are specific to the valid paths of the scoped example. The key difference between TOTEM's approach and ours is that, in ours, all user/title/item creations are carried out as the set-up of each of these 6 testcases, and all user/title/item deletions are carried out as the teardown of each of these 6 testcases. The generation algorithm also generates a test case for each unsuccessful path of each scenario. That is, for unsuccessful paths, there is no need to consider any interleaving or combination of sequences. However, because of how scenarios can be monitored only if they are correctly triggered, our verification tool must allow bypassing a trigger statement for the purpose of testing unsuccessful paths. For example, given that the borrowing use case can only be triggered if the borrowing user has been correctly created, in order to test an attempt to borrow with an invalid user, we must bypass the trigger statement of the borrowing scenario.

Recapitulating, given a scoped example to test (as the very small example considered in TOTEM), we restrict interleaving to only use cases that cannot be shown to be independent of others. For those use cases only, we directly reuse TOTEM's approach to test generation. All object creation and deletion relevant to independent use cases is handled in the setup and teardown shared by all example-specific test cases. A test case is also generated for each unsuccessful paths of each (independent or not) use case. (TOTEM does not address generation for unsuccessful paths of use cases.). In the worst case, there are no independent use cases and our approach boils down to reusing TOTEM's generation algorithm. But, as pointed out by Jacobson (1992), the absence of independent use cases typically indicates an undesirable high level of coupling in the system and should be corrected. Assuming a system will have some independent use cases, eliminating these when interleaving will necessarily generate fewer cases than in TOTEM as it is the length of sequences that correlates to the number of possible interleavings.

At this point of the discussion, given the simplicity of the scoped example, one must consider how cycles (e.g., renewals) may complicate the generation algorithm. On this issue, TOTEM follows Binder's (2000) guideline: each loop or cycle is to be replaced by a fixed number of paths corresponding to at least the minimum and the maximum number of iterations through it. TOTEM does not further discuss this issue and, by default, we adopt their approach. That is, currently, we assume that any use case involved in a cycle is not an independent one and reuse TOTEM's approach to generation of tests to deal with it. We do remark however that future work should investigate this issue further as there is room for improvement. For example, our model does have the ability to verify that if one loan is renewed, all other loans are not affected (i.e., do not exhibit a state change). The open

problem to eventually address is whether such ability could somehow reduce the number of interleavings to generate.

So far, we have claimed that assuming that there are some independent use cases in a system allows us to reduce the number of interleavings to generate (using TOTEM's algorithm) by eliminating these independent use cases from the possible sequences of use cases to consider. But it is crucial to acknowledge that, in our approach, there are more tests to be generated (beyond those associated with unsuccessful paths and interleaving between non-independent use cases). Indeed, additionally, both the setup and the teardown require tests to verify the mutual independence between some use cases, on which our approach rests. Most importantly, the tests for the setup are independent of all others, as are those for the teardown. For the setup, we need tests to verify that a user/title/item creation does not affect other users, nor existing titles, nor existing items, nor existing loans (as explained in 3.2). Similarly, for the teardown, we need tests to verify that a user/title/item deletion does not affect other users, nor existing titles, nor existing items, nor existing loans (again as explained in 3.2). To do so, it is crucial to remark that such tests should not be limited to a particular scoped example. Instead, there should be one testplan for setup and one for the teardown, regardless of the scope used for the testing of non-independent use cases. For our library system, according to our ACL model, we have users, titles, items and loans to consider. It is left to the tester to define how many instances whose states are monitored need to be created. We have arbitrarily chosen 5 as a representative number of instances to consider. The task is *not* to consider all possible interleavings of creating 5 users, 5 titles, and 5 items (which leads to 15! tests). Instead it consists of two steps:

- Step1: minimal set of existing instances

    - Test each of the 3 valid orderings of (A(u1) || I(t1).G(t1,i11)), which addresses creating the first user, the first title, the first item
    - For each of the 3 valid orderings of (A(u1) || I(t1).G(t1,i11))
        - Test A(u2), test I(t2), test G(t1, i12), test G(t2, i21) separately
        - Test the 4! interleavings of A(u2) || I(t2) || G(t1, i12) || G(t2, i21)

- Step 2: representative set of existing instances
    - Once and only once all tests of step 1 have passed, create 5 users, 5 titles, 5 items per title. Add 1 random loan to one user, 2 random loans to another, and 5 random loans to a third user.
    - Test A(u6), test I(t6), test G(t6, i61), test G(t6, i62), test G(t6, i63), test G(t5, i56) and test G(t5, i57) separately

The idea of step 1 is to ensure that, regardless of the order of creation of a set of instances whose states are to remain unchanged, the addition of a new user or a new title, or a new item (to an existing or a new title) indeed does not trigger any incorrect state change. This step proceeds from varying the values used for the different parameters of the independent use cases. Step 2 follows the same idea but once a representative number of instances have been created. Given step 1, we do not consider the ordering of these creations.

For step 1, *for each possible interleaving*, there is small set of tests to perform, which is obtained by testing a new value of each parameter in all possible contexts of use. This explains why we test both adding a new item to an existing title and adding a new item to a new title. For step 2, there is a fixed set of tests, which also proceeds from testing a new value of each parameter in all possible contexts of use.

Quantitatively, for this library system, we end up with 3 + (3*4) + (3*4!) tests for step1, and 7 tests for step 2, for a total of 94 tests for the setup testplan. It should be emphasized that, contrary to TOTEM that relies on combining the current sequence with only one instance of the previous interleavings, here all possible interleavings are considered. That is, our approach appears to be more complete with respect to setup use cases.

For the teardown testplan, we adopt a similar strategy but must consider two additional difficulties: deleting the last instance of some type and deleting a title that has items that have not been deleted.

- Perform A(u1), I(t1), G(t1, i11) (in any order since it is assumed the setup has demonstrated their independence). Test each of the 3 valid orderings of a (C(u1) || (J(t1,i11). H(t1))), which addresses deleting the last instance of a user, a title, an item.
- Perform A(u1), I(t1) and test the 2 orderings of (C(u1) || H(t1)). This addresses the independence of user deletion from title deletion when there are no copies associated with this title.
- Perform A(u1), I(t1), G(t1, i11) and test the 2 orderings of (C(u1) || H(t1)). This addresses the independence of user deletion from title deletion when the latter involves deleting a single copy associated with this title.
- Perform A(u1), I(t1), G(t1, i11), G(t1, i12), G(t1, i13) and test the 2 orderings of (C(u1) || H(t1)). This addresses the independence of user deletion from title deletion when the latter involves deleting several copies associated with this title.

- Perform A(u1), I(t1), G(t1, i11), G(t1, i12), G(t1, i13). Then perform J(t1, i12) and then test the 2 orderings of (C(u1) || H(t1)). This addresses the independence of user deletion from title deletion when the latter involves deleting copies associated with this title before and after user deletion.
- Perform A(u1), I(t1), G(t1, i11), G(t1, i12), G(t1, i13). Test each of the 30 possible orderings of **R**: {C(u1) || [(J(t1,i11) || J(t1,i12) || J(t1,i13)).H(t1)]} (different styles of brackets are used here merely to ease readability). This establishes that the deletion of ANY user is independent of the deletion of multiple copies of a same title.
- Perform A(u2), I(t2), G(t2, i21), G(t2, i22), G(t2, i23) in any order. Previous tests make it irrelevant to consider the interleavings of C(u2) with H(t2) and J(t2, i21), J(t2,i22) and J(t2,i12) given u1 is equivalent to u2, t1 to t2, etc. Perform A(u1), I(t1), G(t1, i11), G(t1, i12), G(t1, i13). Consider the regular expression **R** given in the previous bullet and choose at random one of its 30 possible orderings (since they have been shown to be equivalent in the previous bullet). Call this particular sequence S.
    - Test the 6 orderings given by (C(u2) || S).
    - Test the 6 orderings given by (H(t2) || S).
    - Since they have been shown to be equivalent, choose one of the six possible orderings of J(t2,i21), J(t2,i22), J(t2,i23) followed by H(t2) and test its 126 interleavings with S.

- create 5 users, 5 titles, 5 items per title. Add 1 random loan to one user, 2 random loans to another, and 5 random loans to a third user. Test separately the deletion of each user, item and title.
- create 5 users, 5 titles, 5 items per title. Add 1 random loan to one user, 2 random loans to another, and 5 random loans to a third user. Given interleavings between user, title and item deletions have been tested and deemed equivalent, randomly select and verify a single sequence that tests the deletion of each user, each title and each item.

Despite attempting to be systematic, the test plan for the teardown, much like the one for the setup, is not complete, relying instead on our assumption that it offers sufficient coverage to conclude to the independence of the relevant use cases. The same assumption is made in TOTEM while trying to justify why only 1 sequence of a current set of interleavings is used to further combine with other sequences they have chosen arbitrarily to cover their scoped example… Moreover, TOTEM does not address all deletion sequences we consider. Specifically, it does not consider deleting a title that has still items associated with it.

Should we not tackle completeness and test all possible orderings of the following expression?

(C(u1) || C(u2) || ((J(t1,i11) || J(t1,i12)) || J(t1,i13)
|| J(t2,i21) || J(t2,i22) || J(t2,i23))
. (H(t1) || H(t2))))

Given there are 129,600 such orderings, this is clearly not feasible and demonstrates Binder's (2000) point that the goal of 'complete coverage' is generally impossible to achieve, testers having to rely more than often on their intuition about what constitutes sufficient

coverage. For example, a brute-force approach like tackling the 129,600 interleavings above, ignores completely the relevance of equivalence partitioning. Testing interleavings for a user, a title and copies of this title amounts to testing for any user, any title, and any copy of this title. This is the key observation on which this part of our test generation approach rests.

In total, testing the teardown requires 3+2+2+2+2+30+6+6+126+15+1= 195 tests, and the total scoped example needs 94 (setup) + 6 (specific to the scope example) + 20 (for unsuccessful paths) + 195 (teardown) = 315 tests (in contrast to the 18018 of TOTEM only for valid paths). Furthermore, both the setup and teardown testplans in fact cover more tests than TOTEM has to cover (A || I.J) and (C || J.H) since these testplans are not specific to the scoped example.

4 Discussion and Conclusion

4.1 Lessons learnt

The scoped example that TOTEM investigates in depth is small. However, contrary to that paper, the ACL model we have developed addresses the complete library system. Four lessons proceed from this extensive modeling exercise:

1) Not all of TOTEM's sequential dependencies should be modeled as an ACL scenario. Some are better conceptualized as path conditions relevant to a use case rather than as genuine inter-use-case dependencies. For example, returning all loans before removing a user. This test is not really about a sequence of use cases (as there is a multitude of possible sequences of loans and renewals before returns across several users) but rather about verifying *at a specific point in time* (i.e., when a user is about to be deleted) that no loan exists. This is an important observation. It confirms what was explained earlier, namely that events are *not* sufficient to verify all paths of a use case (especially if loops and cycles are present in a use case): we do need the ability to verify the state of a particular object at a particular point in time, something ACL does support via the notion of an observability. Also, this observation should lead us to question the exact semantic for the edges in the activity diagram of TOTEM (Figure 2): what is exactly a sequential dependency in TOTEM?
2) Several complex dependencies between use cases can be handled in a single ACL scenario, as demonstrated by the last two scenarios of our case study. Again, what must be emphasized is that the semantics of ACL, especially the notion of a parameterized event, allow us to define exact dependencies between use cases involving a *same* object (e.g., of a loan). For example, the parameter ***loanId*** allows us to address the

complexities of successful loan renewal and deletions in a single scenario, ***loanLifeTime***. Since all values for ***loanId*** are considered to be equivalent with respect to testing, scenario ***loanLifeTime*** needs to be exercised with only one actual value for ***loanId*** (and one actual value user id). (All invalid paths are treated in the scenarios corresponding to the use cases.) In turn, this avoids the complex (and arbitrary) algorithm TOTEM uses for determining the values of parameters in use cases (without making a separation between successful and unsuccessful paths of use cases).

3) Because each loan is treated individually with its specific user, its specific title, and its specific item, there is no problem in modeling the same user borrowing a book, later returning it (with or without renewals and fees), and subsequently borrowing it again. However, while modeling this is straightforward, thinking of such a possibility and generating the appropriate test case is not: the problem of test data selection remains a significant challenge.

4) The possible presence of loops in ACL scenarios raises the issue of their coverage. In the TOTEM paper, as previously mentioned, Briand and Labiche (2002) do state that loop coverage can reuse the guidelines put forth by Binder (2000). Their point is that how to test a loop is orthogonal to the issue of sequential dependencies between use cases. As suggested above, we agree. Clearly, it is possible to localize loops inside scenarios in ACL. This is desirable inasmuch as it fits our approach to test generation, namely each scenario must have its possible paths covered by tests. More specifically, a loop can be seen as a set of branches (each corresponding to a specific number of iterations to test) to cover inside a scenario. Binder's recommendation can thus readily apply (e.g., test with 0, 1, average and maximum number of iterations).

Whereas our ACL model tackles the entire library system, our discussion of test generation has focused on the sole example TOTEM presents. Our generation algorithm can be summarized as follows:

A- Test Unsuccessful paths of all scenarios of the ACL model

B- **Separating Independent Use Cases**: From the ACL model, identify all scenarios that rely on state preservation and develop from them the setup and teardown

C- **Testing non-independent use cases**: Reusing TOTEM's generation algorithm on the sequences of non-independent use cases, generate a set of test cases that all use the same setup and the same teardown.

D- **Generate Tests for the Setup**

step 1: test use case independence by creating a minimal set of instances

step 2: test use case independence by creating a representative set of instances

E- Generate Tests for the Test Teardown

step 1: test use case independence by deleting a minimal set of instances

step 2: test use case independence by deleting from a representative set of instances

As Briand and Labiche observe, a mathematical characterization of their generation algorithm (and of ours *a forteriori*) is extremely difficult as it this algorithm depends not only on the number of use cases but also on the number of parameters of each one, on the number of dependencies, and on the parameter dependency between use cases. For example, variations on the value of the parameters for borrowing depend on what users, titles and items have been previously created. Acknowledging this impediment, we have presented a quantitative comparison between TOTEM and our generation algorithm in the context of

this limited example. This preliminary analysis suggests that our approach can significantly reduce the number of generated test cases by separating independent use cases from other before using interleaving with the latter. But, clearly, this claim cannot be generalized before more complex contexts of use of the library system are developed for both TOTEM and our generation algorithm. We must emphasize, however, that the extensive ACL model we have presented is meant to address the complete library system.

4.2 Conclusion and Future Work

The case study we have presented demonstrates it is feasible to model complex sequential dependencies between use cases without necessarily generating an unmanageable number of test cases. But at least three important challenges remain if we are to offer a practical approach to system-level testing using ACL/VF:

1) Scenario monitoring in ACL must be extensively tested, especially its new support for parameterized events.
2) The concept of the state signature of an object is not new and is known to be defeasible. This leads to several future research questions including: a) how to obtain a state signature function that is as indefeasible as possible and b) are state signatures the best approach to verifying state preservation?
3) While preliminary work on automatic generation of test paths out of an ACL model is promising with respect to branch and loop coverage, much remains to be done, especially in terms of the generation test data. Preliminary work in Corriveau's research group suggests ACL must be augmented with TTCN-like capabilities in order to tackle this complex problem.

Bibliography

ACL Specification, http://vf.davearnold.ca/docs/ACLSpec.pdf last accessed in May 2014

Amyot, D., Logrippo, L. and Weiss, M., Generation of test purposes from Use Case Maps. Computer Networks Vol. 49 No. 5: 643-660, 2005

Amyot, D. and Mussbacher, G., User Requirements Notation: The First Ten Years, The Next Ten Years, Journal of Software, Vol. 6, No. 5, 2011

Arnold, D., Corriveau, J.-P. and Shi, W.: Reconciling Offshore Outsourcing with Model-Based Testing, Software Engineering Approaches for Offshore and Outsourced Development (SEAFOOD 2010a), Peterhorf (Saint Petersburg), Russia, pp.6-22, 2010

Arnold, D., Corriveau, J.-P. and Shi, W.: Modeling and Validating Requirements using Executable Contracts and Scenarios, Proceedings of Software Engineering Research, Management & Applications (SERA 2010b), Montreal, Canada, pp.311-320, 2010

Arnold, D., Corriveau, J.-P. and Shi, W.: Scenario Based-Validation: Beyond the User Requirements Notation, Proceedings of the 21st Australian Software Engineering Conference(2010c), Auckland, New Zealand, pp.75-84, 2010

Autili, M., Inverardi, P. and Pelliccione, P., A scenario based notation for specifying temporal properties, Proceeding SCESM '06 Proceedings of the 2006 international workshop on Scenarios and state machines: models, algorithms, and tools, Vol. 1, No. 1, pp.21-28, 2006

Baker, P., Dai, Z.R., Grabowski, J., Schieferdecker, I. and Williams, C., Model-Driven Testing: Using the UML Testing Profile, Springer, 2008

Basanieri, F., Bertolino, A. and Marchetti, E., The Cow_Suite Approach to Planning and Deriving Test Suites in UML Projects, Proc. Fifth Int'l Conf. Unified Modeling Language, Vol. 2460 of LNCS, pp. 383-397, 2002

Bernot, G., Gaudeland, M.-C. and Marre, B., Software Testing Based on Formal Specifications: A Theory and a Tool, Software Eng. J., Vol . 6, No . 6, pp. 387-405, 1991

Binder, R., Testing Object Oriented Systems: Models, Patterns and Tools, Addison-Wesley, 2000

Biswal, B.N., Nanda, P. and Mohapatra, D.P., A Novel Approach for Scenario-Based Test Case Generation, Information Technology (ICIT '08), Bhubaneswar, India, pp.244-247, 2008,

Blueprint, http://www.blueprintsys.com/solution/ last accessed in June 2013

Briand, L. and Labiche, Y., A UML-Based Approach to System Testing, J. Software and Systems Modeling, Vol. 1, No. 1, pp.10-42, 2002

Carver R.H. and Tai, K.-C. , Use of Sequencing Constraints for Specification-Based Testing of Concurrent Programs, IEEE Trans. Software Eng., Vol . 24, No . 6, pp. 471-490, 1998.

Cockburn, A., Structuring Use Cases with Goals, J. Object-Oriented Programming, Vol. 10, No. 5, pp. 35-40 and Vol. 10, No. 7, pp.56-62, 1997

Corriveau, J.-P. and Shi, W., Traceability in Acceptance Testing, Journal of Software Engineering and Applications, (JSEA), Vol.6 No.10A, pp.36-46, 2013

Coudert, J. C., Toward a Symbolic Logic Minimization Algorithm, in Proc. of VLSI Design' 93, Bombay 1993

Dick, J. and Faivre, A., Automating the Generation and Sequencing of Test Cases from Model-based Specifications, Proc. Int'l Symp. Formal Methods Europe, Lecture Notes in Computer Science, Vol . 670, pp. 268-284, 1993

Fröhlich P. and Link, J., Automated Test Case Generation from Dynamic Models, Proc. 14th European Conf. Object-Oriented Programming, Lecture notes in Computer Science ; Vol . 1850, 2000

Fusion, http://c2.com/cgi/wiki?FusionMethodology last accessed in July, 2014

Goëtze, M: UsageTester: UML-oriented Usage Testing. Ilmenau Technical University, Dept. Process Informatics, 2002

Gomaa, H., Designing Concurrent, Distributed, and Real-Time Applications with UML, Addison-Wesley, 2000

Grieskamp, W., Multi-Paradigmatic Model-Based Testing, Technical Report, Microsoft Research, August 2006, pp.1−20

Hartmann, J., Vieira, M., Foster, H. and Ruder, A.: A UML-based approach to system testing, Innovations in System Software Engineering, Vol. 1, No. 1, pp.12-14, 2005

Jacobson, I., Object Oriented Software Engineering, Addison-Wesley, 1992

JavaMop, http://fsl.cs.illinois.edu/index.php/Special:JavaMOP3 last accessed in May 2014

Jena, A.K., Swain, S.K. and Mohapatra, D.P., A novel approach for test case generation from UML activity diagram, Issues and Challenges in Intelligent Computing Techniques (ICICT), 2014 International Conference on, Ghaziabad, India, pp.621-629, 2014

Khandai, M., Abhinna, A., Acharya, A., and Mohapatra, D.P., A Survey on Test Case Generation from UML Model, International Journal of Computer Science and Information Technologies, Vol . 2 No., pp.1164-1171, 2011

Kundu, D. and Samanta, D., A Novel Approach to Generate Test Cases from UML Activity Diagrams, Journal of Object Technology, Vol. 8, No. 3, pp.65-83, 2009

Larsen, K.G., Pettersson, P. and Yi, W., UPPAAL in a nutshell, International Journal on Software Tools for Technology Transfer, Vol. 1 No. 1, pp. 134-152, Springer-Verlag, 1997

Linzhang, W., Jiesong, Y., Xiaofeng, Y., Jun, M., Xuandong, L. and Guoliang, Z., Generating test cases from UML activity diagram based on gray-box method. In 11th Asia-Pacific Software Engineering Conference (APSEC04), pp. 284-291, 2004

Message Sequence Charts, https://www.itu.int/ITU-T/2005-2008/com17/languages/Z120.pdf last accessed in March 2014

Meyer, B. Applying "Design by Contract", in Computer (IEEE), Vol. 25, No. 10, pp. 40–51, 1992

Mingsong, C., Xiaokang, Q. and Xuandong, L., Automatic test case generation for UML activity diagrams. In international workshop on Automation of software test, pp. 2-8, 2006

Nanda, P., Biswal, B.N and Mohapatra, D.P., A Novel Approach for Test Case Generation Using Activity Diagram, International Journal Of Computer Science And Applications Vol. 1, No. 1, pp.60-63, June 2008

Nebut, C., Fleurey, F., Le Traon, Y. and J-M Jezequel, Automatic Test Generation: A Use Case Driven Approach, IEEE Trans. On Software Engineering, Vol. 32, No. 3, MARCH 2006

Offutt J. and Abdurazik, A., Generating Tests from UML Specifications, Proc. Second Int'l Conf. Unified Modeling Language: Beyond the Standard, 1999

Petrinets, http://www.informatik.uni-hamburg.de/TGI/PetriNets/ last accessed in November 2013

Pilone, D., UML 2.0 Pocket Reference, O'Reilly, 2013

Riebisch, M., Philippow, I. and M. Goëtze, "UML-Based Statistical Test Case Generation," Proc. Int'l Conf. Net. Object Days, Vol. 2591, pp. 394-411, 2002

Ryser, J. and Glinz, M., SCENT: A Method Employing Scenarios to Systematically Derive Test Cases for System Test, Universitat Zurich, Institute for Informatics, Zurich

Seifert, D., Helke, S., Santen, T., Test Case Generation for UML Statecharts, Vol 2890 pp. 462-468, PSI'2003

Shirole, M. and Kumar, R., UML behavioral model based test case generation: a survey, ACM SIGSOFT Software Engineering Notes Vol. 38 No. 4, pp.1-13, July 2013

Somé, S.: Specifying Use Case Sequencing Constraints using Description Elements, Sixth International Workshop on Scenarios and State Machines (SCESM'07), p.4, 2007

Spec Explorer, http://visualstudiogallery.msdn.microsoft.com/271d0904-f178-4ce9-956b-d9bfa4902745/ last accessed in May 2014

Sokenou, D., Generating Test Sequences from UML Sequence Diagrams and State Diagrams. IEEE Society, Vol. 2 No. 94, pp. 236-240, 2006

Tripathy, P. and Naik, K., Generation of Adaptive Tests from Nondeterministic Finite State Models, Proceedings of the IFIP TC6/WG6.1 Protocol Test Systems, V, pp. 309-320, 1992

TTCN3, http://www.ttcn-3.org/ last accessed in May 2014

Wirfs-Brock, R. and MacKean, A., Object Design: Roles, Responsibilities, and Collaborations, Addison-Wesley Professional; 2002

Wirfs-Brock, R., The art of writing uses cases, http://www.wirfs-brock.com/PDFs/Art_of_Writing_Use_Cases.pdf last accessed in June 2014

Appendix

This appendix presents the start of the walkthrough of our test plan that consists in test cases for traversing every branch of every scenario of our model.

The complete set of walkthroughs can be found at:

www.scs.carleton.ca/~jeanpier/InamWalkthroughs

The simulation was written to check in particular the firing and observing of parameterized events and the scenario processing for our test plan. These simulations can also be found at above link.

1- Create User u1

Librarian requests creation of a new user and enters the user-information through library terminal. Method which is bound to responsibility **initiateNewUserAddition()** handles the input provided and returns the serialized version of userInfo. The return value of method can be captured by 'value' keyword and stored in contract variable **potentialNewUserInfo**.
potentialNewUserInfo =" JonDoe"
From within this responsibility , a parameterized event **'userAdditionRequested'** is fired with the parameter **potentialNewUserInfo**. Every fired event can be observed elsewhere in the contract. For every unique ***potentialNewUserInfo***, one unique ‘‘userAdditionRequested’ event will be fired. For identification purpose we call the event fired here to be, **'Event1'**.
Event1 - userAdditionRequested("JonDoe")
A new instance of Scenario AddUser gets created. Let call this scenario instance - **'AU1'.**
Value of the parameter in the observed event becomes equal to the parameter value of fired event.
newUInfo="JonDoe"
Scenario grammar will start to execute after this and will verify if the responsibilities and events are getting executed/fired as per the described grammar.
IUT verifies if the ***userInfo*** provided is unique, if so, then a method is called to generate a positive, unique integer to be used as userId. This method is bound to the responsibility **generateNewUserId(String userInfo).** The parameter passed to the method is the same serialized userInfo returned by the first method execution.
Responsibility generateNewUserId(String userInfo) will execute as
generateNewUserId(**"JonDoe"**)

Within Scenario grammar for instance **'AU1' ,** a choice statement, **"choice(DoesUserExist(newUInfo)) true"** is evaluated.
The method bound to ***DoesUserExist*** Observability will execute with parameter - "**JonDoe"**, the method would check for user existence and return **'false'**. So the grammar inside alternative block executes. We assume that integer value return by the IUT method bound to responsibility ***generateNewUserId***() is **1**, which is stored in a scenario variable uId. **uId = 1**
After the userId has been generated, IUT should call a method which is responsible for actually creating the user, which is bound to responsibility **createUser(String userInfo, Integer uId)** and thus its parameters are initialized with the parameter value of the IUT method and the responsibility executes as
createUser("JonDoe", 1)
Within responsibility, after the post conditions are evaluated, a parameterized event **newUserCreated(userInfo, uId)** is fired, parameters userInfo, uId are the parameters of the responsibility ***createUser*** (String userInfo, Integer uId) so the event is fired as **newUserCreated("*JonDoe", 1*)**, and called it '**Event2**'.
Event2 - newUserCreated("JonDoe", 1)
Grammar evaluation of scenario instance **AUI** has been waiting to observe the event **newUserCreated(potentialNewUserInfo, uId)**, so the event expected to be observed is **newUserCreated("JonDoe", 1).** As explained above this event (Event2) was fired from the responsibility createUser() with exactly similar parameters as expected , hence the scenario grammar is satisfied and scenario execution proceeds further and fires an event **userAdditionComplete(uId),** let's call this event as **'Event3'**. This fired event signifies successful completion of the 'create unique user' path.
Event3 - userAdditionCompleted(1)
A new instance **UDUC1** of another scenario **userDeletionFollowsUserCreation is created** when event **Event3** is fired. This scenario observes event **userAdditionCompleted(newUId)** to trigger, so the value of **newUId** is set equal to the parameter with which the event **userAdditionCompleted** was fired. so **newUId = 1**
Now the grammar for this scenario instance **UDUC1** waits to observe **userDeletionCompleted(newUId)** event, so it can terminate. So the Event that it waits to be fired is **userDeletionCompleted(1)**.
Scenario execution for instance **AU1** control exits the alternative() block now and contract level variable **potentialNewUserInfo** is assigned a value of null, to signify that one user creation request processing is completed.
potentialNewUserInfo = null
Scenario instance **AU1** is destroyed upon encountering the keyword Terminate(), a terminate() statement signifies completion of a scenario.

2- Create Title t0

Librarian requests creation of a new title and enters the title-information through library terminal. Method which is bound to responsibility **initiateNewTitleAddition ()** handles the input provided and returns the serialized version(It is assumed that serialized string has comma separated values) of tile information entered by user. The return value of method can be captured by 'value' keyword and stored in contract variable **newTitleInfo**.

Let's assume that title being created has title name **MDD** and its Isbn is **1233**.

newTitleInfo="MDD,1233"

Here a parameterized event **'titleCreationRequested'** is fired with the parameter **newTitleInfo** For identification purpose we call the event fired here to be, **'Event4'**

Event4-titleCreationRequested("MDD,1233")

As this event is fired an instance of scenario addTitle is created. let's call this instance **'AT0'**

Value of parameter variable aTitleInfo in the observe statement becomes equal to the value of the parameter newTitleInfo, with which event 'titleCreationRequested' was fired.

aTitleInfo = " MDD,1233"

Now scenario grammar for instance **'AT0'** will start to execute and will verify if the responsibilities are getting executed as per the scenario grammar.

Choice statement, **"choice(IsTitleInfoComplete(aTitleInfo) true"** is evaluated. The method bound to **IsTitleInfoComplete** Observability will execute with parameter - **" MDD,1233"**, the method would check ,if the given title information is sufficient to create a title, If it is then it would return 'true' otherwise false.

IsTitleInfoComplete(MDD,1233) returns true and the grammar inside choice block executes, where another choice statement "**Choice(DoesTitleExist(aTitleInfo)) false"** is evaluated

The method bound to **DoesTitleExist** Observability will execute with parameter - **"MDD,1233"**, this method checks if the title with the given title information exists, if it does then it would return 'true' otherwise 'false'.

DoesTitleExist("MDD,1233") returns **false**. So the grammar execution enters the choice block, where it waits to observe an event -

newTitleCreated ("MDD,1233")

IUT processes the current request for title creation and should verify , if the given title Information is sufficient for title creation and unique. If so then a method, which takes title information as parameter is called, this method would create the title for the given title information and would be bound to responsibility **createTitle(String titleInfo)**. This responsibility executes as -

createTitle("MDD,1233")

From within the responsibility createTitle, a parameterized event **newTitleCreated (titleInfo)** is fired after the post conditions are evaluated. Parameter **titleInfo** is the parameter of the responsibility **createTitle (String titleInfo**) so the event is fired as **newTitleCreated ("MDD,1233")**. Let's call this event, ***Event5***.

Event5- newTitleCreated ("MDD,1233")
Grammar evaluation of scenario instance **AT0** has been waiting to observe the event **(newTitleCreated("MDD,1233"),** which is in fact **'Event5'.** Scenario execution proceeds further and fires an event **titleAdditionCompleted("MDD,1233"),** let's call this event as **'Event6**. This fired event signifies successful completion of the '**newTitleCreated** ' path.
Event6-titleAdditionCompleted("MDD,1233")
Scenario ***removeTitleFollowsAddTitle***, triggers upon observing the event
titleAdditionComplete("MDD,1233")
and a new instance for this scenario is created, Let's call this instance **RTATRI1**. Value of parameter variable **newTitleInfo** in the observe statement becomes equal to the value of the parameter, with which event **titleAdditionCompleted** was fired.
newTitleInfo = " MDD,1233"
Scenario grammar for instance **RTATRI1** executes further and now waits to observe an event **titleDeletionCompleted ("MDD,1233").**
Scenario execution control for instance **AT0** exits the choice block now and contract level variable **newTitleInfo** is assigned a value of null, to signify that one title creation request processing is completed.
newTitleInfo = null
Scenario instance **AT0** is destroyed upon encountering the keyword Terminate().

3- Delete title t0 (Deleted but never used)

Librarian requests deletion of a title and enters the title-information through library terminal. Method which is bound to responsibility **initiateTitleDeletion ()** handles the input provided and returns the serialized version of tile information entered by user. The return value of method can be captured by 'value' keyword and stored in contract variable **titleInfoForDeleteTitle**.
Let's assume that title being deleted has title name **MDD** and its Isbn is **1233**.
titleInfoForDeleteTitle =" MDD,1233"
Here a parameterized event **titleDeletionRequested** is fired with the parameter **titleInfoForDeleteTitle**. For every unique **titleInfoForDeleteTitle**, one unique **titleDeletionRequested** event will be fired. For identification purpose we call the event fired here to be, **'Event7'**
Event7- titleDeletionRequested ("MDD,1233")
As event **titleDeletionRequested ("MDD,1233")** is fired an instance of scenario **RemoveTitle** is created. let's call this instance **'DT0'**
Value of parameter variable **titleInfoForDeleteTitle** in the observe statement becomes equal to the value of the parameter newTitleInfo, with which event **titleDeletionRequested** was fired.
newInfoForDeletionTitle = "MDD,1233"
Now scenario grammar will start to execute and will verify if the responsibilities are getting executed as per the scenario grammar.

Choice Statement, **"choice(DoesTitleExist(newInfoForDeletionTitle) true"** is evaluated
The method bound to **DoesTitleExist** Observability will execute with parameter - **"MDD,1233"**, the method would check ,if there exists a title with the given title information, If so, then it would return 'true' otherwise false. **DoesTitleExist ("MDD,1233")** returns true and the grammar inside choice block executes.
Scenario execution continues and observability ***FindTitle***(newInfoForDeletionTitle) is called which returns the isbn for the given title information, return value of the observability is stored in scenario variable isbn.
isbn = FindTitle(**"MDD,1233"**)
isbn = 1233
Then the next choice statement **"choice(TitleHasLoans(isbn)) false"** is evaluated.
TitleHasLoans(1233) returns ***false***.
Scenario grammar enters the choice block where a loop is executed in the form of 'each' statement. The 'each' statement iterates through all items in a list, in this case the list is a list of integers (itemIds) , which is returned by observability **GetItems(1233)**.
GetItems(1233) returns a list of all the itemIds for the title with the given isbn. For title with isbn 1233, there is no item at all so the choice loop doesn't execute even once.
Scenario grammar exits the each loop .
Scenario grammar for instance **'DT0'**, now waits to observe an event **titleDeleted(1233)**.

IUT at this point should call a method which would remove the title, this method would take title information as parameter. Responsibility bound to this method is **removeTitle(String titleInfo)**, this responsibility executes as - **removeTitle("MDD,1233")**
From, within this responsibility, after all the post conditions have been verified , an event **titleDeleted("MDD,1233")** is fired **Event8 - titleDeleted("MDD,1233")**
Scenario instance **DT0** was waiting to observe this event **titleDeleted("MDD,1233")** . It observes it and scenario grammar is satisfied and execution continues further. An event **titleDeletionCompleted("MDD,1233")** is fired. Let's call this event Event8.
Event9 - titleDeletionCompleted("MDD,1233")
Scenario grammar for instance **RTATRI1** (of scenario removeTitleFollowsAddTitle) was waiting to observe this event **titleDeletionCompleted("MDD,1233").** It observes this event and scenario grammar executes further.
A terminate statement executes and scenario instance **RTATRI1** is destroyed.
Scenario execution control for instance **DT0** exits the choice block now and contract level variable **titleInfoForDeleteTitle** is assigned a value of null, to signify that one title deletion request processing is completed.
titleInfoForDeleteTitle= null
A terminate statement executes and scenario instance **DT0** is destroyed.

4- Create title t1

Librarian requests creation of a new title and enters the title-information through library terminal. Method which is bound to responsibility **initiateNewTitleAddition ()** handles the input provided and returns the serialized version of tile information entered by user. The return value of method can be captured by 'value' keyword and stored in contract variable **newTitleInfo**.
Let's assume that title being created has title name "**Moby Dick**" and its Isbn is **1234**.
newTitleInfo="Moby Dick,1234"
Here a parameterized event **'titleCreationRequested'** is fired with the parameter **newTitleInfo** For identification purpose we call the event fired here to be, **'Event10'**
Event10-titleCreationRequested("Moby Dick,1234")
As this event is fired an instance of scenario addTitle is created. let's call this instance **'AT1'**
Value of parameter variable ***'aTitleInfo'*** in the observe statement becomes equal to the value of the parameter ***'newTitleInfo'***, with which event 'titleCreationRequested' was fired.
aTitleInfo = "Moby Dick,1234"
Now scenario grammar for instance **'AT1'** will start to execute and will verify if the responsibilities are getting executed as per the scenario grammar.
Choice statement, **"choice(IsTitleInfoComplete(aTitleInfo) true"** is evaluated. The method bound to **IsTitleInfoComplete** Observability will execute with parameter - **"Moby Dick,1234"**, the method would check ,if the given title information is sufficient to create a title, If it is then it would return 'true' otherwise false.
IsTitleInfoComplete("Moby Dick,1234") returns true and the grammar inside choice block executes. **IsTitleInfoComplete("Moby Dick,1234")** returns true and the grammar inside choice block executes, where another choice statement
"**Choice(DoesTitleExist(aTitleInfo)) false"** is evaluated
The method bound to **DoesTitleExist** Observability will execute with parameter -**"Moby Dick,1234"**, this method checks if the title with the given title information exists, if it does then it would return 'true' otherwise 'false'.
DoesTitleExist("Moby Dick,1234") returns **false**. So the grammar execution enters the choice block, where it waits to observe an event - **newTitleCreated ("Moby Dick,1234")**
IUT processes the current request for title creation and should verify , if the given title Information is sufficient for title creation and unique. If so then a method, which takes title information as parameter is called, this method would create the title for the given title information and would be bound to responsibility **createTitle(String titleInfo)**. This responsibility executes as -
createTitle("Moby Dick,1234")
From within the responsibility createTitle, a parameterized event **newTitleCreated (titleInfo)** is fired after the post conditions are evaluated. Parameter **titleInfo** is the parameter of the responsibility **createTitle (String titleInfo**) so the event is fired as **newTitleCreated ("Moby Dick,1234")**. Let's call this event, **Event11**.
Event11- newTitleCreated ("Moby Dick,1234")

Grammar evaluation of scenario instance **AT1** has been waiting to observe the event **(newTitleCreated("Moby Dick,1234"),** which is in fact **'Event11'.** Scenario execution proceeds further and fires an event **titleAdditionCompleted("Moby Dick,1234"),** let's call this event as **'Event12'**. This fired event signifies successful completion of the '**newTitleCreated** ' path.

Event12-titleAdditionComplete("Moby Dick,1234")

Scenario removeTitleFollowsAddTitle, triggers upon observing the event **titleAdditionComplete("Moby Dick,1234")**

and a new instance for this scenario is created, . Value of parameter variable **newTitleInfo** in the observe statement becomes equal to the value of the parameter, with which event **titleAdditionCompleted** was fired.

newTitleInfo = "Moby Dick,1234"

Scenario grammar for instance **RTATRI2** executes further and now waits to observe an event **titleDeletionCompleted ("Moby Dick,1234").**

Scenario execution control for instance for instance **'AT1'** exits the choice block now and contract level variable **newTitleInfo** is assigned a value of null, to signify that one title creation request processing is completed. **newTitleInfo = null**

Scenario instance **AT1** is destroyed upon encountering the keyword Terminate().

5- Create copy c1 of t1

Librarian requests creation of a new item and enters the title-information through library terminal. Method which is bound to responsibility **initiateNewItemAddition()** handles the input provided and returns the serialized version of title information entered by librarian. The return value of method can be captured by 'value' keyword and stored in contract variable **titleInfoForAddItem**.

titleInfoForAddItem ="Moby Dick,1234"

Here a parameterized event **itemAdditionRequested** is fired with the parameter **titleInfoForAddItem**. For identification purpose we call the event fired here to be, **'Event13**

Event13- itemAdditionRequested ("Moby Dick,1234")

As this event is fired an instance of scenario **itemAddition** is created. let's called this instance **'IA1T1'**

Value of parameter newTitleInfoForAddItem becomes equal to the value of parameter, with which the event was fired-

newTitleInfoForAddItem = ("Moby Dick,1234")

Now scenario grammar for instance **'IA1T1'** will start to execute -

A choice statement, "choice(DoesTitleExist(newTitleInfoForAddItem)) true" is evaluated

DoesTitleExist("Moby Dick,1234") returns true. Grammar inside choice block executes. Two belief statements are executed, which report that "Title info should be displayed on

terminal" and "Existing items for this title should be listed on the library terminal". Scenario grammar exits the choice block and continues the execution.

FindTitle Observability is called, method bound to this Observability returns the isbn value for given title information which will be stored in the scenario variable isbn.

isbn = FindTitle("Moby Dick,1234")

isbn=1234

IUT processes the current request for item creation and it should verify if there is a title for the given tile information (title, to which item has to be added) and if there is, then method responsible for adding an item to the title will execute. This method would return itemId for the item that is created and added to the title, let's assume that for this request, item that gets created has itemId 1 . A responsibility **addItem(Integer isbn)** is bound to this IUT method ,which executes as -

addItem(1234)

from within the responsibility addItem, a parameterized event **newItemAdded(isbn, value)** is fired and the parameters are isbn and the return value itemId of responsibility. So the event is fired as **newItemAdded(1234,1)** let's call this event as '**Event14**'.

Event14- newItemAdded("Moby Dick,1234",1)

Scenario grammar for scenario instance **'IA1T1'**, expects **addItem(1234)** responsibility to be executed and captures the return value of such execution in a variable - itemId. As explained above responsibility **addItem(1234)** has executed with exactly the same parameter as expected so the scenario grammar is satisfied.

itemId = 1

An event **newItemAdded("Moby Dick,1234",1)** is expected to be observed, by grammar for scenario instance **'IA1T1'** which is event 13, so scenario grammar satisfies and execution continues.

Scenario instance **'IA1T1'** is destroyed upon encountering the keyword Terminate(). Upon termination one more event itemAdditionCompleted(itemId) is fired to signify item addition, for a particular itemId. Let's call it ***Event 15***

Event15- itemAdditionCompleted(1)

A scenario instance **'IL1'** is created for scenario **itemLifeTime** upon fire of event ***itemAdditionCompleted***. Value of aItem is set equal to the parameter with which the event **itemAdditionCompleted** was fired. **aItem = 1**

Inside choice statement of scenario instance **IL1**, the method bound to **DoesItemExist** Observability will execute with parameter - **aItem . DoesItemExist(1)** returns true. So ,Grammar inside choice block executes. Another choice statement **Choice(IsItemAvailable(1)) true** evaluates, **IsItemAvailable(1)** returns true. So ,Grammar inside choice block executes. The grammar scenario now waits to observe event **itemBorrowed(1) OR itemDeletionCompleted(1) .**

A new instance **RI1T1** for scenario removeItemfollowsAddItem is also created when event ***itemAdditionCompleted(1)*** is fired. Value of parameter newItemId in the observed event becomes equal to the parameter of the fired event - ***newItemId*** = 1

Scenario grammar execution for instance **RI1T1**, continues further and now waits for event itemDeletionCompleted(1) to be fired.

6- Create user u2

Librarian requests creation of a new user and enters the user-information through library terminal. Method which is bound to responsibility **initiateNewUserAddition()** handles the input provided and returns the serialized version of userInfo. The return value of method can be captured by 'value' keyword and stored in contract variable **potentialNewUserInfo**.
potentialNewUserInfo =" JaneDoe"
From within this responsibility , a parameterized event **'userAdditionRequested'** is fired with the parameter **potentialNewUserInfo**. For identification purpose we call the event fired here to be, **'Event16'**.
Event16 - userAdditionRequested("JaneDoe")
A new instance of Scenario AddUser gets created. Let call this scenario instance - **'*AU2*'.**
Value of the parameter in the observed event becomes equal to the parameter value of fired event.
newUInfo="JaneDoe"
Scenario grammar execution starts after the scenario is triggered.
IUT verifies if the userInfo provided is unique, if so, then a method is called to generate a positive, unique integer to be used as userId. This method is bound to the responsibility **generateNewUserId(String userInfo).** The parameter passed to the method is the same serialized userInfo returned by the first method execution.
Responsibility generateNewUserId(String userInfo) will execute as
generateNewUserId("JaneDoe").
Within Scenario grammar for instance **'*AU2*'** a choice statement,
"choice(DoesUserExist(newUInfo)) true" is evaluated.
The method bound to **DoesUserExist** Observability will execute with parameter - "**JaneDoe"**, the method would check for user existence and return 'false'. So the grammar inside alternative block will execute. We assume that integer value return by the IUT method bound to responsibility generateNewUserId() is 2, which is stored in a scenario variable uId. **uId = 2**
After the userId has been generated, IUT should call a method which is responsible for actually creating the user, which is bound to responsibility **createUser(String userInfo, Integer uId)** and thus its parameters are initialized with the parameter value of the IUT method and the responsibility executes as
createUser("JaneDoe", 2)
Within responsibility, after the post conditions are evaluated, a parameterized event **newUserCreated(userInfo, uId)** is fired. Parameters userInfo, uId are the parameters of the responsibility createUser (String userInfo, Integer uId) so the event is fired as newUserCreated(**"JaneDoe", 2**), and called it '**Event17**.

Event17 - newUserCreated("JaneDoe", 2)
Grammar evaluation of scenario instance '***AU2***' has been waiting to observe the event **newUserCreated(potentialNewUserInfo, uId)**, so the event expected to be observed is **newUserCreated("JaneDoe", 2).** As explained above this event (**Event17**) was fired from the responsibility ***createUser***() with exactly similar parameters as expected , hence the scenario grammar is satisfied and scenario execution proceeds further and fires an event **userAdditionComplete(uId),** let's call this event as **'Event18'**. This fired event signifies successful completion of the 'create unique user' path.
Event18 - userAdditionCompleted(2)
A new instance **UDUC2** of another scenario **userDeletionFollowsUserCreation** is created when event **Event18** is fired. This scenario observes event **userAdditionCompleted(newUId)** to trigger, so the value of **newUId** is set equal to the parameter with which the event **userAdditionCompleted** was fired.
newUId = 2
Now the grammar for this scenario instance **UDUC2** waits to observe **userDeletionCompleted(newUId)** event, so it can terminate. So the Event that it waits to be fired is **userDeletionCompleted(2)**.
Scenario execution for instance **AU2** control exits the alternative() block now and contract level variable **potentialNewUserInfo** is assigned a value of null, to signify that one user creation request processing is completed.
potentialNewUserInfo = null
Scenario instance **AU2** is destroyed upon encountering the keyword Terminate().

7- Create copy c2 of t1

Librarian requests creation of a new item and enters the title-information through library terminal. Method which is bound to responsibility **initiateNewItemAddition()** handles the input provided and returns the serialized version of title information entered by librarian. The return value of method can be captured by 'value' keyword and stored in contract variable **titleInfoForAddItem**.
titleInfoForAddItem ="Moby Dick,1234**"**
Here a parameterized event **itemAdditionRequested** is fired with the parameter **titleInfoForAddItem**. For identification purpose we call the event fired here to be, **'Event19'**
Event19- itemAdditionRequested ("Moby Dick,1234")
As this event is fired an instance of scenario itemAddition is created. let's called this instance **'IA2T1'**
Value of parameters **newTitleInfoForAddItem** becomes equal to the value of parameter with which event is fired.
newTitleInfoForAddItem= "**Moby Dick,1234"**
Now scenario grammar for instance **'IA2T1'** will start to execute -

A choice statement, "choice(DoesTitleExist(newTitleInfoForAddItem)) true" is evaluated
DoesTitleExist("Moby Dick,1234") returns true. Grammar inside choice block executes. Two belief statements are executed, which report that "Title info should be displayed on terminal" and "Existing items for this title should be listed on the library terminal". Scenario grammar exits the choice block and continues the execution.
FindTitle Observability is called, method bound to this Observability returns the isbn value for given title information which will be stored in the scenario variable isbn.
isbn = FindTitle("Moby Dick,1234")
isbn=1234
IUT processes the current request for item creation and should verify if there is a title for the given tile information (title, to which item has to be added) and if there is, then method responsible for adding an item to the title will execute. This method would return itemId for the item that is created and added to the title, let's assume that for this request, item that gets created has itemId **2 .** A responsibility **addItem(Integer isbn)** is bound to this IUT method ,which executes as
addItem(1234)
From within the responsibility addItem, a parameterized event **newItemAdded(isbn, value)** is fired and the parameters are isbn and the return value itemId of responsibility. So the event is fired as **newItemAdded(1234,2)** let's call this event as '**Event20**'.
Event20- newItemAdded("**Moby Dick,1234",2)**
Within the choice block, scenario grammar for scenario instance **'IA2T2'**, expects addItem(1234) responsibility to be executed and captures the return value of such execution in a variable - itemId.
As explained above responsibility addItem(1234) has executed with exactly the same parameter as expected so the scenario grammar is satisfied. **itemId = 2**
An event **newItemAdded("Moby Dick,1234",2)** is expected to be observed, by grammar for scenario instance **'IA2T1'** which is ***Event20***, so scenario grammar satisfies and execution continues.
Scenario instance **'IA2T1'** is destroyed upon encountering the keyword Terminate(). Upon termination one more event itemAdditionCompleted(itemId) is fired to signify item addition, for a particular itemId. Let's call it **Event21'**
Event21- itemAdditionCompleted(2)
A scenario instance **'IL2'** is created for scenario **itemLifeTime** upon fire of event itemAdditionCompleted. Value of aItem is set equal to the parameter with which the event **itemAdditionCompleted** was fired. **aItem = 2**
Inside choice statement of scenario instance **IL2**, the method bound to **DoesItemExist** Observability will execute with parameter - **aItem . DoesItemExist(2)** returns true. So ,Grammar inside choice block executes. Another choice statement **Choice(IsItemAvailable(2)) true** evaluates, **IsItemAvailable(2)** returns true. So ,Grammar inside choice block executes. The grammar scenario now waits to observe event **itemBorrowed(2) OR itemDeletionCompleted(2) .**

A new instance RI2T1 for scenario ***removeItemfollowsAddItem*** is also created when event ***itemAdditionCompleted(2)*** is fired. Value of parameter newItemId in the observed event becomes equal to the parameter of the fired event - ***newItemId = 2***
Scenario grammar execution for instance RI2T1, continues further and now waits for event itemDeletionCompleted(2) to be fired.

8- User u1 borrows c1 of t1 First Borrow

Librarian requests creation of a new loan and enters the user and item information through library terminal. Method which is bound to responsibility initiateNewLoanCreation (Integer userId, Integer itemId) handles the input provided.
Responsibility initiateNewLoanCreation executes as -
initiateNewLoanCreation(1,1)
Within responsibility initiateNewLoanCreation, value of parameters userId, itemId is stored in contract variables loanRequestedForUserId, loanRequestedForItemId.
loanRequestedForUserId = 1
loanRequestedForItemId = 1
After responsibility's post conditions are evaluated, a parameterized event **loanRequested** is fired with two parameters with values of contract variables **loanRequestedForUserId**, **loanRequestedForItemId**. Let's call this event **'Event22'**.
Event22-**loanRequested(1, 1)**
A new Scenario instance **BLAUAI1** for scenario borrowLoanFollowsAddUserAndAddItem is created when event **loanRequested(1, 1)** is fired. Values of variables newUserId, newItemId are set as -
newUserId = 1
newItemId = 1
Next, the scenario grammar for instance **BLAUAI1** executes a choice statement **"Choice(DoesUserExist(newUserId)) true". DoesUserExist(1)** returns true, where one more choice statement **"choice(DoesItemExist(newItemId))true"** is evaluated **DoesItemExist(1)** returns true. Scenario grammar for instance **BLAUAI1** now waits to observe an event **loanAdded(1,1)**.
Scenario borrowLoanCopy also triggers, by observing the event **loanRequested(1, 1)**
newLoanRequestedForUserId, **newLoanRequestedForItemId** are scenario variables, whose value is set equal to the parameter values with which the event **loanRequested** was fired. A new scenario instance is created, let's call this instance - **BLC1**.
newLoanRequestedForUserId = 1
newLoanRequestedForItemId = 1
Scenario grammar execution proceeds and a choice statement "**choice(FindUserInfo(newLoanRequestedForUserId) not= null) true"**, is evaluated.
Observability **FindUserInfo(1)** executes and returns userInfo as a string - "**JonDoe**".
Grammar execution enters the choice block -

Observability **GetUserLoanCount(newLoanRequestedForUserId)** is executed as **GetUserLoanCount(1)**, which returns 0 because there are no current loans for the given userId. The return value is stored in a scenario variable userLoanCount.
userLoanCount = 0
Observability GetTitleInfoForItem(newLoanRequestedForItemId) is executed as **GetTitleInfoForItem(1)**, which returns **"Moby Dick,1234"** string as response. The return value is stored in a scenario variable titleInfo.
titleInfo = "Moby Dick,1234"
Scenario grammar execution continues and a series of nested choice/alternative blocks are executed, which are placed to verify that responsibility to add loan is executed, only if user is eligible for a loan.
First Choice Statement - "**choice(userLoanCount < Parameters.MaxLoanCount) true"**
Value of Parameter MaxLoanCount was set at the time of ACL-IUT binding to 3. hence 0<3 statement evaluates to true.
Second Choice Statement **- choice(FindTitle(titleInfo) > 0) true**
Observability **FindTitle(Moby Dick,1234)** executes and returns **Isbn** for the given title information as 1234. hence 1234 > 0 evaluates to true.
Third Choice Statement - "**Choice(IsItemAvailable(newLoanRequestedForItemId)) true"**
Observability **IsItemAvailable (1)** executes and returns true.
fourth Choice Statement -
"**Choice(IsItemAReferenceCopy(newLoanRequestedForItemId)) false"**
Observability **IsItemAReferenceCopy(1)** executes and returns false, since this item is not a reference item.
fifth Choice Statement "**choice(UserHasPrivilege(newLoanRequestedForUserId)) false"**

Observability **UserHasPrivilege(1)** executes and returns true, since this user does have privilege. scenario grammar execution for instance **BLC1** jumps the choice block and now waits to observe an event **loanAdded(1,1)**.
Upon receiving the request for borrowing an item for a particular user, IUT should verify if the user is eligible for getting a loan and execute a method, which would add a loan for the given user for the given itemId and return the loanId for the loan that was added. Post execution It would also increase the number current loan count for this user. Parameters passed to this method would be userId =1 , itemID = 1. Responsibility bound to this method is addLoan(String userId, String itemId),which would execute as - **addLoan(1,1)**.
The return value for this responsibility is integer value **100** (LoanId has been given a random value to track loans).
Within the responsibility, an event **loanAdded(userId, itemId)** is fired after all the post conditions are evaluated. Let's call this ***Event23***.
Event23- **loanAdded(1, 1)**

Scenario **loanLifeTime** triggers upon observing this event and a new instance is created, Let's call this scenario instance **LLT1.** Parameters of the triggered event for scenario instance **LLT1** are set as

fineUId = 1

fineItemId = 1

Scenario grammar execution for instance **LLT1** continues. An Observability **FindLoan(1,1)** is called , which returns the loanId for the given parameters. The return value is stored in scenario variable fineLoanId.

fineLoanId = 100

Next, a choice statement **choice(fineLoanId >0) true** is evaluated and the grammar execution enters the choice block. Now scenario instance **LLT1** waits to observe any one of the three events - **loanExpired(1,1)) loanRenewedCompleted(1,1) loanDeletionCompleted(1,1)**

Scenario grammar for instance **BLAUAI1** has also been waiting to observe this event ***loanAdded***(1, 1), which now can be observed and scenario grammar is satisfied.

Terminate() statement is executed next, which causes the scenario instance **BLAUAI1** to be destroyed.

scenario grammar for instance **BLC1**, was waiting for an loanAdded(1,1). Event24 **loanAdded(1, 1)** satisfies the scenario grammar and grammar execution continues for instance **BLC1**.

An event **itemBorrowed(newLoanRequestedForItemId)** is fired to signify that a particular item has been borrowed and not available for loan any more. let's call this event **Event24**.

Event24- itemBorrowed(1)

A scenario instance **IL1** was waiting to observe one of the two events **itemBorrowed(1) or itemDeletionCompleted(1)** .**itemBorrowed(1)** happens first and scenario grammar is satisfied. Next, a choice statement **choice(DoesItemExist(1)) true** is evaluated , **DoesItemExist(1)** returns true, so the grammar inside choice block executes. Further, one more choice statement **Choice(IsItemAvailable(1)) false** is evaluated. **IsItemAvailable(1)** returns false, so the grammar inside choice block, executes. Scenario grammar for instance **IL1**, now waits to observe an event **itemReturned(1)** .

Scenario grammar for instance **BLC1**, now evaluates a choice statement "**choice((userLoanCount+1) == Parameters.MaxLoanCount) true"**, to verify, if user now has taken maximum number of allowed loans.

Parameter MaxLoanCount was initialized at the time of ACL-IUT binding to the value of 3.

((0+1) == 3) evaluates to false.

Scenario grammar execution exits the choice block.

Contract variable **loanRequestedForUserId**, **loanRequestedForItemId** are assigned a value 0 to signify that current borrow item request for an item for a particular user has been completed.

loanRequestedForUserId = 0

loanRequestedForItemId = 0
Scenario terminates and scenario instance **BLC1** is destroyed.

9- Create title t2

Librarian requests creation of a new title and enters the title-information through library terminal. Method which is bound to responsibility **initiateNewTitleAddition ()** handles the input provided and returns the serialized version(It is assumed that serialized string has comma separated values) of title information enter by user. The return value of method can be captured by 'value' keyword and stored in contract variable **newTitleInfo**.
Let's assume that title being created has title name **ModelTesting** and its Isbn is **1235**
newTitleInfo="ModelTesting,1235"
Here a parameterized event **'titleCreationRequested'** is fired with the parameter **newTitleInfo**. For identification purpose we call the event fired here to be, **'Event25'**
Event25-titleCreationRequested("ModelTesting,1235")
As this event is fired an instance of scenario addTitle is created. let's call this instance **'AT2'**
Value of parameter variable aTitleInfo in the observe statement becomes equal to the value of the parameter newTitleInfo, with which event 'titleCreationRequested' was fired.
aTitleInfo = "ModelTesting,1235"
Now scenario grammar will start to execute and will verify if the responsibilities are getting executed as per the scenario grammar.
Choice statement, "choice(IsTitleInfoComplete(aTitleInfo) true" is evaluated. The method bound to **IsTitleInfoComplete** Observability will execute with parameter - **"ModelTesting,1235"**, the method would check ,if the given title information is sufficient to create a title, If it is then it would return 'true' otherwise false.
IsTitleInfoComplete(ModelTesting,1235) returns true and the grammar inside choice block executes. , where another choice statement **Choice(DoesTitleExist(aTitleInfo))** is evaluated
The method bound to **DoesTitleExist** Observability will execute with parameter - **"ModelTesting,1235"**, this method checks if the title with the given title information exists, if it does then it would return 'true' otherwise 'false'.
DoesTitleExist("ModelTesting,1235") returns true. So the grammar execution enters the choice block, where it waits to observe an event -
newTitleCreated ("ModelTesting,1235")

IUT processes the current request for title creation and should verify , if the given title Information is sufficient for title creation and unique. If so then a method, which takes title information as parameter is called, this method would create the title for the given

title information and would be bound to responsibility **createTitle(String titleInfo)**. This responsibility executes as -
createTitle("ModelTesting,1235")
From within the responsibility createTitle, a parameterized event **newTitleCreated (titleInfo)** is fired after the post conditions are evaluated. Parameter **titleInfo** is the parameter of the responsibility **createTitle (String titleInfo**) so the event is fired as **newTitleCreated ("ModelTesting,1235")**. Let's call this event, **Event26**.
Event26- newTitleCreated ("ModelTesting,1235")
Grammar evaluation of scenario instance **AT2** has been waiting to observe the event **(newTitleCreated("ModelTesting,1235"),** which is in fact **Event26.** Scenario execution proceeds further and fires an event **titleAdditionCompleted("ModelTesting,1235"),** let's call this event as **'Event27'**. This fired event signifies successful completion of the '**newTitleCreated** ' path.
'Event27'-titleAdditionCompleted("ModelTesting,1235")

Scenario removeTitleFollowsAddTitle, triggers upon observing the event
titleAdditionCompleted("Moby Dick,1234")
and a new instance for this scenario is created, let's call this instance **RTATRI3**. Value of parameter variable **newTitleInfo** in the observe statement becomes equal to the value of the parameter, with which event **titleAdditionCompleted** was fired.
newTitleInfo = "ModelTesting,1235"
Scenario grammar for instance **RTATRI3** executes further and now waits to observe an event **titleDeletionCompleted ("ModelTesting,1235").**

Scenario execution for instance **AT2** control exits the choice block now and contract level variable **newTitleInfo** is assigned a value of null, to signify that one title creation request processing is completed.
newTitleInfo = null
Scenario instance **AT2** is destroyed upon encountering the keyword Terminate(), a terminate() statement signifies completion of a scenario.

10- Create copy c1 of t2

This is assumed that only information required at the time of item creation is title-info.
-Librarian requests creation of a new item and enters the title-information through library terminal. Method which is bound to responsibility **initiateNewItemAddition()** handles the input provided and returns the serialized version of title information entered by librarian. The return value of method can be captured by 'value' keyword and stored in contract variable **titleInfoForAddItem**.
titleInfoForAddItem ="ModelTesting,1235"

Here a parameterized event **itemAdditionRequested** is fired with the parameter **titleInfoForAddItem**. For identification purpose we call the event fired here to be, **'Event28'**

Event28- itemAdditionRequested ("ModelTesting,1235")

As event is fired an instance of scenario itemAddition is created. let's called this instance **'IA1T2'**

Value of parameter newTitleInfoForAddItem becomes equal to the value of parameter, with which the event was fired-

newTitleInfoForAddItem = ("ModelTesting,1235")

Now scenario grammar for instance **'IA1T2'** will start to execute -

A choice statement, "choice(DoesTitleExist(newTitleInfoForAddItem)) true" is evaluated

DoesTitleExist("ModelTesting,1235") returns true. Grammar inside choice block executes. Two belief statements are executed, which report that "Title info should be displayed on terminal" and "Existing items for this title should be listed on the library terminal". Scenario grammar exits the choice block and continues the execution.

FindTitle Observability is called, method bound to this Observability returns the isbn value for given title information which will be stored in the scenario variable isbn.

isbn = FindTitle("ModelTesting,1235")

isbn=1235

IUT processes the current request for item creation and should verify if there is a title for the given tile information (title, to which item has to be added) and if there is, then method responsible for adding an item to the title will execute. This method would return itemId for the item that is created and added to the title, let's assume that for this request, item that gets created has **itemId 4** . A responsibility **addItem(Integer isbn)** is bound to this IUT method ,which executes as

addItem(1235)

From within the responsibility addItem, a parameterized event **newItemAdded(isbn, value)** is fired and the parameters are isbn and the return value itemid of responsibility. So the event is fired as **newItemAdded(1235,4)** let's call this event as '**Event29**'.

'Event29'- newItemAdded("ModelTesting,1235",4)

Scenario grammar for scenario instance **'IA1T2'**, expects **addItem(1235)** responsibility to be executed and captures the return value of such execution in a variable - itemId. As explained above responsibility **addItem(1235)** has executed with exactly the same parameter as expected so the scenario grammar is satisfied.

itemId = 4

An event **newItemAdded**(**"ModelTesting,1235", 4**) is expected to be observed, which is **'Event29'**, so scenario grammar satisfies and execution continues.

Scenario instance **'IA1T2'** is destroyed upon encountering the keyword Terminate(). Upon termination one more event itemAdditionCompleted(itemId) is fired to signify item addition, for a particular itemId. Let's call it **Event30**

Event30- itemAdditionCompleted(4)

A scenario instance '**IL4**' is created for scenario **itemLifeTime** upon fire of event itemAdditionCompleted. Value of aItem is set equal to the parameter with which the event **itemAdditionCompleted** was fired.
aItem = 4
Inside choice statement of scenario instance '**IL4**', the method bound to **DoesItemExist** Observability will execute with parameter - **aItem . DoesItemExist(4)** returns true. So ,Grammar inside choice block executes. Another choice statement **Choice(IsItemAvailable(4)) true** evaluates, **IsItemAvailable(4)** returns true. So ,Grammar inside choice block executes. The grammar scenario now waits to observe event **itemBorrowed(4) OR itemDeletionCompleted(4) .**
A new instance ***RI1T2*** for scenario ***removeItemfollowsAddItem*** is also created when event ***itemAdditionCompleted(4)*** is fired. Value of parameter newItemId in the observed event becomes equal to the parameter of the fired event - newItemId = 4
Scenario grammar execution for instance ***RI1T2***, continues further and now waits for event itemDeletionCompleted(4) to be fired.

11- User u2 borrows c2 of t1 Second Borrow

Librarian requests creation of a new loan and enters the user and item information through library terminal. Method which is bound to responsibility initiateNewLoanCreation (Integer userId, Integer itemId) handles the input provided. Responsibility initiateNewLoanCreation executes as -
initiateNewLoanCreation(2,2)
Within responsibility initiateNewLoanCreation, value of parameters userId, itemId is stored in contract variables loanRequestedForUserId, loanRequestedForItemId.
loanRequestedForUserId = 2
loanRequestedForItemId = 2
After responsibility's post conditions are evaluated, a parameterized event **loanRequested** is fired with two parameters with values of contract variables **loanRequestedForUserId**, **loanRequestedForItemId**. Let's call this event **'Event31'**.
'Event31'-loanRequested(2, 2)
A new Scenario instance **BLAUAI2** borrowLoanFollowsAddUserAndAddItem for scenario borrowLoanFollowsAddUserAndAddItem is created when event **loanRequested(2, 2)** is fired. Values of variables newUserId, newItemId are set as -
newUserId = 2
newItemId = 2
Next, the scenario grammar for instance **BLAUAI2** executes a choice statement "**Choice(DoesUserExist(newUserId)) true ". DoesUserExist(2)** returns true, where one more choice statement "**choice(DoesItemExist(newItemId))true"** is evaluated. **DoesItemExist(2)** returns true. Scenario grammar for instance **BLAUAI2** now waits to observe an event **loanAdded(2, 2)**.

Scenario borrowLoanCopy also triggers, by observing the event **loanRequested(2,2)** **newLoanRequestedForUserId**, **newLoanRequestedForItemId** are scenario variables, whose value is set equal to the parameter values with which the event **loanRequested** was fired. A new scenario instance is created, let's call this instance - **BLC2**.
newLoanRequestedForUserId = 2
newLoanRequestedForItemId = 2
Scenario grammar execution proceeds and a choice statement **choice(FindUserInfo(newLoanRequestedForUserId) not= null) true**, is evaluated.
Observability **FindUserInfo(2)** executes and returns userInfo as a string - "**JaneDoe**".
Grammar execution enters the choice block -
Observability **GetUserLoanCount(newLoanRequestedForUserId)** is executed as **GetUserLoanCount(2)**, which returns 0 because there are no current loans for the given userId. The return value is stored in a scenario variable userLoanCount.
userLoanCount = 0
Observability GetTitleInfoForItem(newLoanRequestedForItemId) is executed as **GetTitleInfoForItem(2)**, which returns **"Moby Dick,1234"** string as response. The return value is stored in a scenario variable titleInfo.
titleInfo = "Moby Dick,1234"
Scenario grammar execution continues and a series of nested choice/alternative blocks are executed, which are placed to verify that responsibility to add loan is executed, only if user is eligible for a loan.
First Choice Statement - **choice(userLoanCount < Parameters.MaxLoanCount) true**
Value of Parameter MaxLoanCount was set at time of ACL-IUT binding to 3. hence 0<3 statement evaluates to true.
Second **Choice Statement - choice(FindTitle(titleInfo) > 0) true**
Observability **FindTitle(Moby Dick,1234)** executes and returns **Isbn** for the given title information as 1234. hence 1234 > 0 evaluates to true.
Third Choice Statement - **Choice(IsItemAvailable(newLoanRequestedForItemId)) true**
Observability **IsItemAvailable (2)** executes and returns true.
Fourth Choice Statement -
Choice(IsItemAReferenceCopy(newLoanRequestedForItemId)) false
Observability **IsItemAReferenceCopy(2)** executes and returns false, since this item is not a reference item.
Fifth Choice Statement - **choice(UserHasPrivilege(newLoanRequestedForUserId)) false**

Observability **UserHasPrivilege(2)** executes and returns true, since this user does have privilege. scenario grammar execution for instance **BLC2** jumps the choice block and now waits to observe an event **loanAdded(2, 2)**.
Upon receiving the request for borrowing an item for a particular user, IUT should verify if the user is eligible for getting a loan and execute a method, which would add a loan for the given user for the given itemId and return the loanId for the loan that was added. Post execution It would also increase the number current loan count for this

user. Parameters passed to this method would be **userId =2 , itemID = 2** Responsibility bound to this method is addLoan(String userId, String itemId), which would execute as - **addLoan(2,2)**.

The return value for this responsibility is integer value **101** (LoanId has been given a random value to track loans).

Within the responsibility, an event **loanAdded(userId, itemId)** is fired after all the post conditions are evaluated. Let's call this event **Event32**.

Event32- **loanAdded(2, 2)**

Scenario **loanLifeTime** triggers upon observing this event and a new instance is created, Let's call this scenario instance **LLT2.**

Parameters of the triggered event for scenario instance **LLT1** are set as

fineUId = 2

fineItemId = 2

Scenario grammar execution for instance **LLT2** continues. An Observability **FindLoan(2,2)** is called , which returns the loanId for the given parameters. The return value is stored in scenario variable fineLoanId.

fineLoanId = 101

Next, a choice statement **choice(fineLoanId >0) true** is evaluated and the grammar execution enters the choice block. Now scenario instance **LLT2** waits to observe any one of the three events - **loanExpired(2,2) loanRenewedCompleted(2,2) loanDeletionCompleted(2,2)**

Scenario grammar for instance **BLAUAI2** has been waiting to observe this event loanAdded(**2, 2**), which now can be observed and scenario grammar is satisfied. Terminate() statement is executed next, which causes the scenario instance **BLAUAI2** to be destroyed.

scenario grammar for instance **BLC2**, was waiting for an **loanAdded(2,2)**. **Event34**- **loanAdded(2,2)** satisfies the scenario grammar and grammar execution continues for instance **BLC2**.

An event **itemBorrowed(newLoanRequestedForItemId)** is fired to signify that a particular item has been borrowed and not available for loan any more. let's call this event **Event33**.

Event33- itemBorrowed(2)

A scenario instance **IL2** was waiting to observe one of the two events **itemBorrowed(2) or itemDeletionCompleted(2)** .**itemBorrowed(2)** happens first and scenario grammar is satisfied. Next, a choice statement **choice(DoesItemExist(2)) true** is evaluated , **DoesItemExist(2)** returns true, so the grammar inside choice block executes. Further, one more choice statement **Choice(IsItemAvailable(2)) false** is evaluated. **IsItemAvailable(2)** returns false, so the grammar inside choice block, executes. Scenario grammar for instance **IL2**, now waits to observe an event **itemReturned(2)** .

Scenario grammar for instance **BLC1**, now evaluates a choice statement "**choice((userLoanCount+1) == Parameters.MaxLoanCount) true"**, to verify if user now has taken maximum number of allowed loans.
Parameter MaxLoanCount was initialized at the time of ACL-IUT binding to the value of 3. **((0+1) == 3) evaluates to false**.
Scenario grammar execution exits the choice block.
Contract variable **loanRequestedForUserId**, **loanRequestedForItemId** are assigned a value 0 to signify that current borrow item request for an item for a particular user has been completed. **loanRequestedForUserId = 0**
loanRequestedForItemId = 0
Scenario terminates and scenario instance **BLC2** is destroyed.

12- Create copy c3 of t1

Librarian requests creation of a new item and enters the title-information through library terminal. Method which is bound to responsibility **initiateNewItemAddition()** handles the input provided and returns the serialized version of title information entered by librarian. The return value of method can be captured by 'value' keyword and stored in contract variable **titleInfoForAddItem**.
titleInfoForAddItem ="Moby Dick,1234"
Here a parameterized event **itemAdditionRequested** is fired with the parameter **titleInfoForAddItem**. For identification purpose we call the event fired here to be, **'Event34'**
'Event34'- itemAdditionRequested ("Moby Dick,1234")
As this event is fired an instance of scenario itemAddition is created. let's called this instance **'IA3T1'**
Value of parameter newTitleInfoForAddItem becomes equal to the value of parameter, with which the event was fired-
newTitleInfoForAddItem = ("Moby Dick,1234")
Now scenario grammar r for instance **'IA3T1'** will start to execute -
A choice statement, "choice(DoesTitleExist(newTitleInfoForAddItem)) true" is evaluated
DoesTitleExist("Moby Dick,1234") returns true. Grammar inside choice block executes. Two belief statements are executed, which report that "Title info should be displayed on terminal" and "Existing items for this title should be listed on the library terminal".
Scenario grammar exits the choice block and continues the execution.
FindTitle Observability is called, method bound to this Observability returns the isbn value for given title information which will be stored in the scenario variable isbn.
isbn = FindTitle("Moby Dick,1234")
isbn=1234
IUT processes the current request for item creation and should verify if there is a title for the given tile information (title, to which item has to be added) and if there is, then

method responsible for adding an item to the title will execute. This method would return itemId for the item that is created and added to the title, let's assume that for this request, item that gets created has **itemId 3** . A responsibility **addItem(Integer isbn)** is bound to this IUT method ,which executes as
addItem(1234)
from within the responsibility addItem, a parameterized event **newItemAdded(isbn, value)** is fired and the parameters are isbn and the return value itemid of responsibility. So the event is fired as **newItemAdded(1234,3)** let's call this event as **'Event35'**.
Event35- newItemAdded("Moby Dick,1234",3)
Within the choice block, scenario grammar for scenario instance **'IA3T1'**, expects addItem(**1234**) responsibility to be executed and captures the return value of such execution in a variable - itemId.
As explained above responsibility addItem(**1234**) has executed with exactly the same parameter as expected so the scenario grammar is satisfied.
itemId = 3
An event newItemAdded("**Moby Dick,1234**",**3**) is expected to be observed, which is event **Event35**, so scenario grammar satisfies and execution continues.
Scenario instance **'IA3T1'**is destroyed upon encountering the keyword Terminate().
Upon termination one more event itemAdditionCompleted(itemId) is fired to signify item addition, for a particular itemId. Let's call it **Event35'**
Event35- itemAdditionCompleted(3)
A scenario instance **IL3** is created for scenario **itemLifeTime** upon fire of event itemAdditionCompleted. Value of aItem is set equal to the parameter with which the event **itemAdditionCompleted** was fired.
aItem = 3
Inside choice statement of scenario instance **IL3**, the method bound to **DoesItemExist** Observability will execute with parameter - **aItem . DoesItemExist(3)** returns true. So ,Grammar inside choice block executes. Another choice statement **Choice(IsItemAvailable(3)) true** evaluates, **IsItemAvailable(3)** returns true. So ,Grammar inside choice block executes. The grammar scenario now waits to observe event **itemBorrowed(3) OR itemDeletionCompleted(3) .**
A new instance RI3T1 for scenario ***removeItemfollowsAddItem*** is also created when event ***itemAdditionCompleted(3)*** is fired. Value of parameter newItemId in the observed event becomes equal to the parameter of the fired event - ***newItemId*** = 3
Scenario grammar execution for instance RI3T1, continues further and now waits for event ***itemDeletionCompleted(3)*** to be fired.

13- Create copy c2 of t2

Librarian requests creation of a new item and enters the title-information through library terminal. Method which is bound to responsibility **initiateNewItemAddition()**

handles the input provided and returns the serialized version of title information entered by librarian. The return value of method can be captured by 'value' keyword and stored in contract variable **titleInfoForAddItem**.

titleInfoForAddItem ="ModelTesting,1235"

Here a parameterized event **itemAdditionRequested** is fired with the parameter **titleInfoForAddItem**. For identification purpose we call the event fired here to be, **'Event36'**

'Event36- itemAdditionRequested ("ModelTesting,1235")

As this event is fired an instance of scenario itemAddition is created. let's called this instance **'IA2T2'**

Value of parameter newTitleInfoForAddItem becomes equal to the value of parameter, with which the event was fired-

newTitleInfoForAddItem = ("ModelTesting,1235")

Now scenario grammar for instance **'IA2T2'** will start to execute -

A choice statement, "choice(DoesTitleExist(newTitleInfoForAddItem)) true" is evaluated

DoesTitleExist("ModelTesting,1235") returns true. Grammar inside choice block executes. Two belief statements are executed, which report that "Title info should be displayed on terminal" and "Existing items for this title should be listed on the library terminal". Scenario grammar exits the choice block and continues the execution.

FindTitle Observability is called, method bound to this Observability returns the isbn value for given title information which will be stored in the scenario variable isbn.

isbn = FindTitle("ModelTesting,1235")

isbn=1235

IUT processes the current request for item creation and should verify if there is a title for the given tile information (title, to which item has to be added) and if there is, then method responsible for adding an item to the title will execute. This method would return itemId for the item that is created and added to the title, let's assume that for this request, item that gets created has **itemId 5** . A responsibility **addItem(Integer isbn)** is bound to this IUT method ,which executes as

addItem(1235)

from within the responsibility addItem, a parameterized event **newItemAdded(isbn, value)** is fired and the parameters are isbn and the return value itemid of responsibility. So the event is fired as **newItemAdded(1234,5)** let's call this event as **'Event37'**.

'Event37'- newItemAdded("ModelTesting,1235,5")

Within the choice block, scenario grammar for scenario instance **'IA2T2'**, expects addItem(**1235**) responsibility to be executed and captures the return value of such execution in a variable - itemId.

As explained above responsibility **addItem(1235)** has executed with exactly the same parameter as expected so the scenario grammar is satisfied.

itemId = 5

An event **newItemAdded("ModelTesting,1235,5")** is expected to be observed, by grammar for scenario instance **'IA1T2'** which is ***event37***, so scenario grammar satisfies and execution continues.
Scenario instance **'IA2T2'** is destroyed upon encountering the keyword Terminate(). Upon termination one more event itemAdditionCompleted(itemId) is fired to signify item addition, for a particular itemId. Let's call it **Event38**
Event38- itemAdditionCompleted(5)
A scenario instance **'IL5'** is created for scenario **itemLifeTime** upon fire of event itemAdditionCompleted. Value of aItem is set equal to the parameter with which the event **itemAdditionCompleted** was fired.
aItem = 5
Inside choice statement of scenario instance **IL5**, the method bound to **DoesItemExist** Observability will execute with parameter - **aItem . DoesItemExist(5)** returns true. So ,Grammar inside choice block executes. Another choice statement **Choice(IsItemAvailable(5)) true** evaluates, **IsItemAvailable(5)** returns true. So ,Grammar inside choice block executes. The grammar scenario now waits to observe event **itemBorrowed(5) OR itemDeletionCompleted(5) .**
A new instance ***RI2T2*** for scenario ***removeItemfollowsAddItem*** is also created when event ***itemAdditionCompleted(5)*** is fired. Value of parameter newItemId in the observed event becomes equal to the parameter of the fired event -
newItemId = 5
Scenario grammar execution for instance ***RI2T12*** continues further and now waits for event ***itemDeletionCompleted(5)*** to be fired.

14- Delete c3 of t1 Deleted but never used

Librarian requests deletion of an item and enters the title-information through library terminal whose item he wants to delete.
The method bound to responsibility **requestItemDeletion()** handles the incoming item deletion request. It returns the serialized form of the information entered. The returned title information is stored in contract variable **titleInfoForDeleteItem**.
titleInfoForDeleteItem="Moby Dick,1234"
After post conditions are evaluated then a parameterized event **itemDeletionRequestedForATitle** is fired with parameter of value of contract variable **titleInfoForDeleteItem**. Let's call this event as **Event39**.
Event39- itemDeletionRequestedForATitle("Moby Dick,1234")
Now scenario removeItem is triggered by observing this event
itemDeletionRequestedForATitle(titleInfoForDeleteItem)
The value of **newTitleInfoForDeleteItem** is set equal to the parameter value with which event is fired. A scenario instance is created. Let's call it **DL3T1**
newTitleInfoForDeleteItem= " Moby Dick,1234"

Scenario grammar proceeds and a choice statement "**choice(DoesTitleExist(newTitleInfoForDeleteItem)) true"** is evaluated. **DoesTitleExist** Observability will execute with parameter - **DoesTitleExist(" Moby Dick,1234").** **DoesTitleExist(" Moby Dick,1234")** returns true**.** Grammar inside choice block executes.
IUT should verify if there exist a title for the given title information and call a method, which would handle the librarian input for choosing an item to be deleted from the title information displayed on the terminal. This method would return itemId of the item selected. This method is bound to a responsibility returnSelectedItemIdForDeletion(newTitleInfoForDeleteItem). Responsibility returnSelectedItemIdForDeletion executes as -
returnSelectedItemIdForDeletion(" Moby Dick,1234")
Within responsibility , return value can be captured by 'value' keyword and stored in contract variable
itemIdToBeDeleted=3
After post conditions are evaluated then a parameterized event **itemDeletionRequested(itemIdToBeDeleted)** is fired **.** Let's call this event **Event40**
Event40-itemDeletionRequested(3)
Scenario grammar for instance **DL3T1** proceeds and expects the responsibility **returnSelectedItemIdForDeletion(newTitleInfoForDeleteItem)** to be executed, value of newTitleInfoForDeleteItem is "**Moby Dick,1234**" **.** As explained above the same responsibility has executed with the expected parameter, hence scenario grammar is satisfied. Return value of responsibility **returnSelectedItemIdForDeletion()** is stored in a scenario variable newItemIdToBeDeleted.
newItemIdToBeDeleted = 3
A choice statement "**choice(IsItemAvailable(newItemIdToBeDeleted)) true"** is evaluated. **IsItemAvailable(3)** returns true. Grammar inside choice block executes.
Scenario grammar for instance **DL3T1** waits to observe the event itemDeleted(newItemIdToBeDeleted), so the event expected to be observed is **itemDeleted(3).**
Within IUT it should be verified, if the item to be deleted is available or not, If it is, then a method should be called, which would actually delete the item . This method would take the itemId as parameter. Responsibility removeItem(Integer itemId) is bound to this method so it executes as -
removeItem(3)
Within the responsibility, after all post conditions have been verified, an event ***itemDeleted(itemId)*** is fired Let's call this event as **Event41**
Event41 - itemDeleted(3)
The same event Event41 was expected to be observed in scenario instance **DL3T1**, so the scenario is satisfied and scenario grammar execution continues. Another event **itemDeletionCompleted(newItemIdToBeDeleted)** is fired and. Let's call this **Event42**.
Event42 - itemDeletionCompleted(3)

Scenario instance **RI3T1** for scenario removeItemfollowsAddItem has been waiting to observe this event **itemDeletionCompleted(3).** This event is observed and scenario grammar of instance **RI3T1** is satisfied. Next, a Terminate statement is executed and instance **RI3T1** is destroyed.
Scenario instance **IL3** for scenario itemLifeTime has been waiting to observe one of the two events **itemDeletionCompleted(3)** OR **itemBorrowed(3)**
Since ***itemDeletionCompleted(3)*** is fired first, scenario grammar for instance **IL3** is satisfied and grammar execution will proceed further. A choice statement **choice(DoesItemExist(3)) true** is evaluated. DoesItemExist(**3**) returns false so choice block is skipped. Next, a **redo** statement is encountered, which is for the outer choice block **Choice(DoesItemExist(aItem)) true** , the redo statement causes the execution control to go back to the start of the related choice statement. DoesItemExist(**3**) returns false so this entire choice block is skipped. Next, the **Terminate()** statement executes and scenario instance **IL3** is destroyed.

Scenario grammar for instance **DL3T1** execution control exits two choice() blocks and contract level variables itemIdToBeDeleted is assigned zero and titleInfoForDeleteItem is assigned a value of null, to signify that item deletion processing has completed.
itemIdToBeDeleted = 0;
titleInfoForDeleteItem = null
Scenario instance **DL3T1** is destroyed upon encountering the keyword Terminate(), a terminate() statement signifies completion of a scenario.

15- User u1 borrows c2 of t2 Third Borrow

Librarian requests creation of a new loan and enters the user and item information through library terminal. Method which is bound to responsibility initiateNewLoanCreation (Integer userId, Integer itemId) handles the input provided. Responsibility initiateNewLoanCreation executes as -
initiateNewLoanCreation(1,5)
Within responsibility initiateNewLoanCreation, value of parameters userId, itemId is stored in contract variables loanRequestedForUserId, loanRequestedForItemId.
loanRequestedForUserId = 1
loanRequestedForItemId = 5
After responsibility's post conditions are evaluated, a parameterized event **loanRequested** is fired with two parameters with values of contract variables **loanRequestedForUserId**, **loanRequestedForItemId**. Let's call this event **'Event43'**.
'Event43-loanRequested(1, 5)
A new Scenario instance **BLAUAI3** borrowLoanFollowsAddUserAndAddItem for scenario borrowLoanFollowsAddUserAndAddItem is created when event **loanRequested(1, 5)** is fired**.** Values of variables newUserId, newItemId are set as -
newUserId = 1

newItemId = 5
Next, the scenario grammar for instance **BLAUAI3** executes a choice statement **Choice(DoesUserExist(newUserId)) true DoesUserExist(2)** returns true, where one more choice statement **choice(DoesItemExist(newItemId))true** is evaluated **DoesItemExist(5)** returns true. Scenario grammar for instance **BLAUAI3** now waits to observe an event **loanAdded(1, 5)**.

Scenario borrowLoanCopy triggers, by observing the event **loanRequested(1, 5)** **newLoanRequestedForUserId**, **newLoanRequestedForItemId** are scenario variables, whose value is set equal to the parameter values with which the event **loanRequested** was fired. A new scenario instance is created; let's call this instance - **BLC3**.
newLoanRequestedForUserId = 1
newLoanRequestedForItemId = 5
Scenario grammar execution proceeds and a choice statement **choice(FindUserInfo(newLoanRequestedForUserId) not= null) true**, is evaluated.
Observability **FindUserInfo(1)** executes and returns userInfo as a string - "**JonDoe**".
Grammar execution enters the choice block -
Observability **GetUserLoanCount(newLoanRequestedForUserId)** is executed as **GetUserLoanCount(1)**, which returns **1** because there is **one** current loan for the given userId. The return value is stored in a scenario variable userLoanCount.
userLoanCount = 1
Observability **GetTitleInfoForItem(newLoanRequestedForItemId)** is executed as **GetTitleInfoForItem(5)**, which returns **("ModelTesting,1235")** string as response. The return value is stored in a scenario variable titleInfo.
titleInfo = ("ModelTesting,1235")
Scenario grammar execution continues and a series of nested choice/alternative blocks are executed, which are placed to verify that responsibility to add loan is executed, only if user is eligible for a loan.
First Choice Statement - **choice(userLoanCount < Parameters.MaxLoanCount) true**
Value of Parameter MaxLoanCount was set at the time of ACL-IUT binding to 3. hence **1<3** statement evaluates to true.
Second **Choice Statement - choice(FindTitle(titleInfo) > 0) true**
Observability **FindTitle(("ModelTesting,1235")** executes and returns **Isbn** for the given title information as **1235**. hence 1235 > 0 evaluates to true.
Third Choice Statement - **Choice(IsItemAvailable(newLoanRequestedForItemId)) true**
Observability **IsItemAvailable (5)** executes and returns true.
Fourth Choice Statement -
Choice(IsItemAReferenceCopy(newLoanRequestedForItemId)) false
Observability **IsItemAReferenceCopy(5)** executes and returns false, since this item is not a reference item.

Fifth Choice Statement - **choice(UserHasPrivilege(newLoanRequestedForUserId)) false** Observability **UserHasPrivilege(1)** executes and returns true, since this user does have privilege. , scenario grammar execution for instance **BLC3** jumps the choice block and now waits to observe an event **loanAdded(1,5)**.

Upon receiving the request for borrowing an item for a particular user, IUT should verify if the user is eligible for getting a loan and execute a method, which would add a loan for the given user for the given itemId and return the loanId for the loan that was added. Post execution It would also increase the number current loan count for this user. Parameters passed to this method would be **userId =1 , itemID = 5** Responsibility bound to this method is addLoan(String userId, String itemId), which would execute as - **addLoan(1,5)**.

The return value for this responsibility is integer value **102** (LoanId has been given a random value to track loans).

Within the responsibility, an event **loanAdded(userId, itemId)** is fired after all the post conditions are evaluated. Let's call this event **Event44**.

Event44- **loanAdded(1, 5)**

Scenario **loanLifeTime** triggers upon observing this event and a new instance is created, Let's call this scenario instance **LLT3.**

Parameters of the triggered event for scenario instance **LLT1** are set as

fineUId = 1

fineItemId = 5

Scenario grammar execution for instance **LLT3** continues. An Observability **FindLoan(1,5)** is called , which returns the loanId for the given parameters. The return value is stored in scenario variable fineLoanId.

fineLoanId = 102

Next, a choice statement **choice(fineLoanId >0) true** is evaluated and the grammar execution enters the choice block. Now scenario instance **LLT3** waits to observe any one of the three events - **loanExpired(1, 5) loanRenewedCompleted(1, 5) loanDeletionCompleted(1, 5)**

Scenario grammar for instance **BLAUAI3** has been waiting to observe this event ***loanAdded***(**1, 5**), which now can be observed and scenario grammar is satisfied. Terminate() statement is executed next, which causes the scenario instance **BLAUAI3** to be destroyed.

scenario grammar for instance **BLC3**, was waiting for an event **loanAdded(1,5)**. ***Event44-*** **loanAdded(1, 5)** satisfies the scenario grammar and grammar execution continues for instance **BLC3**.

An event **itemBorrowed(newLoanRequestedForItemId)** is fired to signify that a particular item has been borrowed and not available for loan any more. let's call this event **Event45**.

Event45- itemBorrowed(5)

A scenario instance **IL5** was waiting to observe one of the two events **itemBorrowed(5) or itemDeletionCompleted(5)** .**itemBorrowed(5)** happens first and scenario grammar is satisfied. Next, a choice statement **choice(DoesItemExist(5)) true** is evaluated , **DoesItemExist(5)** returns true, so the grammar inside choice block executes. Further, one more choice statement **Choice(IsItemAvailable(5)) false** is evaluated. **IsItemAvailable(5)** returns false, so the grammar inside choice block, executes. Scenario grammar for instance **IL5**, now waits to observe an event **itemReturned(5)** .
Scenario grammar for instance **BLC1**, now evaluates a choice statement **choice((userLoanCount+1) == Parameters.MaxLoanCount) true**, is evaluated to verify if user now has taken maximum number of allowed loans.
Parameter MaxLoanCount was initialized at the time of ACL-IUT binding to the value of 3.
((1+1) == 3) evaluates to false.
Scenario grammar execution exits the choice block.
Contract variable **loanRequestedForUserId**, **loanRequestedForItemId** are assigned a value 0 to signify that current borrow item request for an item for a particular user has been completed.
loanRequestedForUserId = 0
loanRequestedForItemId = 0
Scenario terminates and scenario instance **BLC3** is destroyed.

16- User u2 borrows c2 of t1 Fourth Borrow

Librarian requests creation of a new loan and enters the user and item information through library terminal. Method which is bound to responsibility initiateNewLoanCreation (Integer userId, Integer itemId) handles the input provided.
Responsibility initiateNewLoanCreation executes as -
initiateNewLoanCreation(2,2)
Within responsibility initiateNewLoanCreation, value of parameters userId, itemId is stored in contract variables loanRequestedForUserId, loanRequestedForItemId.
loanRequestedForUserId = 2
loanRequestedForItemId = 2
After responsibility's post conditions are evaluated, a parameterized event **loanRequested** is fired with two parameters with values of contract variables **loanRequestedForUserId**, **loanRequestedForItemId**. Let's call this event **'Event46'**.
'Event46'-loanRequested(2, 2)
A new Scenario instance **BLAUAI4** for scenario borrowLoanFollowsAddUserAndAddItem is created when event **loanRequested(2, 2)** is fired. Values of variables newUserId, newItemId are set as -
newUserId = 2
newItemId = 2

Next, the scenario grammar for instance **BLAUAI4** executes a choice statement "**Choice(DoesUserExist(newUserId)) true". DoesUserExist(2)** returns true, where one more choice statement **choice(DoesItemExist(newItemId))true** is evaluated. **DoesItemExist(2)** returns true. Scenario grammar for instance **BLAUAI4** now waits to observe an event
loanAdded(2, 2).
Scenario borrowLoanCopy also triggers, by observing the event **loanRequested(2, 2). newLoanRequestedForUserId**, **newLoanRequestedForItemId** are scenario variables, whose value is set equal to the parameter values with which the event **loanRequested** was fired. A new scenario instance is created, let's call this instance - **BLC4**.
newLoanRequestedForUserId = 2
newLoanRequestedForItemId = 2
Scenario grammar execution proceeds and a choice statement **"choice(FindUserInfo(newLoanRequestedForUserId) not= null) true"**, is evaluated. Observability **FindUserInfo(2)** executes and returns userInfo as a string - "**JaneDoe**". Grammar execution enters the choice block -
Observability **GetUserLoanCount(newLoanRequestedForUserId)** is executed as **GetUserLoanCount(2)**, which returns **1** because there is one current loan for the given userId. The return value is stored in a scenario variable userLoanCount.
userLoanCount = 1
Observability GetTitleInfoForItem(newLoanRequestedForItemId) is executed as **GetTitleInfoForItem(2)**, which returns **"Moby Dick,1234"** string as response. The return value is stored in a scenario variable titleInfo.
titleInfo = "Moby Dick,1234"
Scenario grammar execution continues and a series of nested choice/alternative blocks are executed, which are placed to verify that responsibility to add loan is executed, only if user is eligible for a loan.
First Choice Statement - **choice(userLoanCount < Parameters.MaxLoanCount) true**
Value of Parameter MaxLoanCount was set at the time of ACL-IUT binding to 3. hence **1<3** statement evaluates to true.
Second "**Choice Statement - choice(FindTitle(titleInfo) > 0) true"**
Observability **FindTitle(Moby Dick,1234)** executes and returns **Isbn** for the given title information as 1234. hence 1234 > 0 evaluates to true.
Third Choice Statement - "**Choice(IsItemAvailable(newLoanRequestedForItemId)) true"**
Observability **IsItemAvailable (2)** executes and returns **false**, because this items has been already borrowed by the same user and has not been returned yet.
So Grammar execution goes to alternative block for third choice statement, which has a belief that is placed there, which says that "librarian should be informed through library terminal that item has been loaned to some other user".
Scenario grammar will now exit the alterative block.

Contract variable **loanRequestedForUserId**, **loanRequestedForItemId** are assigned a value 0 to signify that current borrow item request for an item for a particular user has been completed.
loanRequestedForUserId = 0
loanRequestedForItemId = 0
Scenario terminates and scenario instance **BLC4** is destroyed.

17- Attempt to delete c2 of t1 Copy in use

Librarian requests deletion of an item and enters the title-information through library terminal whose item he wants to delete.
The method bound to responsibility **requestItemDeletion()** handles the incoming item deletion request. It returns the serialized form of the information entered. The returned title information is stored in contract variable **titleInfoForDeleteItem**.
titleInfoForDeleteItem="Moby Dick,1234"
After post conditions are evaluated then a parameterized event **itemDeletionRequestedForATitle** is fired with parameter of value of contract variable **titleInfoForDeleteItem**. Let's call this event as **Event47**.
Event47- itemDeletionRequestedForATitle("Moby Dick,1234")
Now scenario removeItem triggers by observing this event
itemDeletionRequestedForATitle(titleInfoForDeleteItem)
The value of **newTitleInfoForDeleteItem** is set equal to the parameter value with which event is fired. A scenario instance is created. Let's call it **ADL2T1**
newTitleInfoForDeleteItem= "Moby Dick,1234"
Scenario grammar proceeds and a choice statement "**choice(DoesTitleExist(newTitleInfoForDeleteItem)) true"** is evaluated. **DoesTitleExist** Observability will execute with parameter - **DoesTitleExist("Moby Dick,1234") . DoesTitleExist("Moby Dick,1234")** returns true**.** Grammar inside choice block executes. IUT should verify if there exist a title for the given title information and call a method, which would handle the librarian input for choosing an item to be deleted from the title information displayed on the terminal. This method would return itemId of the item selected. This method is bound to a responsibility r**eturnSelectedItemIdForDeletion(newTitleInfoForDeleteItem)**. Responsibility returnSelectedItemIdForDeletion executes as -
returnSelectedItemIdForDeletion("Moby Dick,1234")
Within responsibility , return value can be captured by 'value' keyword and stored in contract variable
itemIdToBeDeleted=2
After post conditions are evaluated then a parameterized event **itemDeletionRequested(itemIdToBeDeleted)** is fired **.** Let's call this event **Event48**

Event48-itemDeletionRequested(2)
Scenario instance **RTATRI2** from scenario **removeTitleFollowsAddTitleAndRemoveItem** was expecting this responsibility to execute, return value of this responsibility is stored in scenario variable itemId. **itemId = 2**
Next, **RTATRI2** waits to observe **itemDeletionCompleted(2).**
Scenario grammar for instance **ADL2T1** proceeds and expects the responsibility **returnSelectedItemIdForDeletion(newTitleInfoForDeleteItem)** to be executed, value of newTitleInfoForDeleteItem is "Moby Dick,1234" **.** As explained above the same responsibility has executed with the expected parameter, hence scenario grammar is satisfied. Return value of responsibility **returnSelectedItemIdForDeletion()** is stored in a scenario variable newItemIdToBeDeleted.
newItemIdToBeDeleted = 2
A choice statement "**choice(IsItemAvailable(newItemIdToBeDeleted)) true"** is evaluated. **IsItemAvailable(2)** returns **False**. Scenario grammar execution for instance **ADL2T1** will move to alternate block where a belief states that librarian should get a message that this item is on loan and can't be deleted.
Scenario grammar for instance **ADL2T1** execution control exits alternative block and contract level variables itemIdToBeDeleted is assigned zero and titleInfoForDeleteItem is assigned a value of null, to signify that item deletion processing has completed.
itemIdToBeDeleted = 0;
titleInfoForDeleteItem = null
Scenario instance **ADL2T1** is destroyed upon encountering the keyword Terminate(), a terminate() statement signifies completion of a scenario.

18- Attempt to delete title t1

Librarian requests deletion of a title and enters the title-information through library terminal. Method which is bound to responsibility **initiateTitleDeletion ()** handles the input provided and returns the serialized version of tile information entered by user. The return value of method can be captured by 'value' keyword and stored in contract variable **titleInfoForDeleteTitle**.
Let's assume that title being deleted has title name **Moby Dick** and its Isbn is **1234**.
titleInfoForDeleteTitle ="Moby Dick,1234"
Here a parameterized event **titleDeletionRequested** is fired with the parameter **titleInfoForDeleteTitle**. For identification purpose we call the event fired here to be, **'Event49'**
Event49- titleDeletionRequested ("Moby Dick,1234")
As event **titleDeletionRequested ("Moby Dick,1234")** is fired an instance of scenario **RemoveTitle** is created. let's call this instance **'DT1'**
Value of parameter variable **titleInfoForDeleteTitle** in the observe statement becomes equal to the value of the parameter newTitleInfo, with which event **titleDeletionRequested** was fired.

newInfoForDeletionTitle = "Moby Dick,1234"
Now scenario grammar will start to execute and will verify if the responsibilities are getting executed as per the scenario grammar.
Choice Statement, **"choice(DoesTitleExist(newInfoForDeletionTitle) true"** is evaluated The method bound to **DoesTitleExist** Observability will execute with parameter - **"Moby Dick,1234"**, the method would check ,if there exists a title with the given title information, If so, then it would return 'true' otherwise false. **DoesTitleExist ("Moby Dick,1234")** returns true and the grammar inside choice block executes.
Scenario execution continues and observability ***FindTitle***(newInfoForDeletionTitle) is called which returns the isbn for the given title information, return value of the observability is stored in scenario variable isbn.
isbn = FindTitle(**"Moby Dick,1234"**) Then the next choice statement **"choice(TitleHasLoans(isbn)) false"** is evaluated. **TitleHasLoans(1234)** returns ***true.*** because title has item on loan so the alternate statement is executed, where a belief statement is executed, which reports that "librarian should be informed that this title has loan so it can't be deleted".
Scenario execution control exits the alternative() block now and contract level variable **titleInfoForDeleteTitle** is assigned a value of null, to signify that one title deletion request processing is completed.
titleInfoForDeleteTitle= null
Scenario instance **DT1** is destroyed upon encountering the keyword Terminate().

19- User u2 renews c2 of t1 First Renewal, loan has not expired

Librarian requests to renew a loan and enters the user and item information through library terminal. The method bound to responsibility **initiateRenewLoan(Integer userId, Integer itemId)** handles the input provided. Responsibility **initiateRenewLoan** executes as
initiateRenewLoan (2, 2)
for this responsibility the value of parameters (userId, itemId) are stored in contract variable **renewLoanRequestedForUserId**
and **renewLoanRequestedForItemId**.
renewLoanRequestedForUserId =2
renewLoanRequestedForItemId =2
After responsibility's post conditions are evaluated, a parameterized event **renewLoanRequested (renewLoanRequestedForUserId, renewLoanRequestedForItemId) is fired.**
Let's call this event as **Event50**.
Event50- renewLoanRequested(2,2)
Scenario renewLoanFollowsBorrowLoan is triggered on observing the event **renewLoanRequested(2,2)** and a new scenario instance is created, let's call this instance - **RLBL1**

loanUserId and **loanItemId** are scenario variables, whose value is set as -
loanUserId =2
loanItemId =2
Scenario grammar executes further for instance **RLBL1** and an Observability **FindLoan(2,2)** executes and it's return value is stored in a scenario variable renewLoanId. Method bound to Observability **FindLoan(2,2)** would return a positive loanId if there exist a loan for itemId 2 for a user with userId 2 otherwise it returns 0. **FindLoan (2,2)** returns integer value **101**.
renewLoanId = 101
Scenario grammar for instance **RLBL1** executes further and a choice statement "**choice(renewLoanId > 0)) true"** is evaluated**.** Grammar execution enters the choice block, where another choice statement "**Choice(IsLoanOverdue(renewLoanId)) true"** is evaluated. Observability **IsLoanOverdue(101)** returns a Boolean value of true, if the loan has expired otherwise False. **IsLoanOverdue(101)** returns False. So the grammar execution goes to the alternative block. Scenario grammar for instance **RLBL1** now waits to observe an event **loanRenewedCompleted (2,2).**

Scenario **renewLoan** is also triggered on observing the event **renewLoanRequested(2,2)** (Event54) and a new scenario instance is created, let's call this instance - **RENL1**
newRenewLoanRequestedForUserId and **newRenewLoanRequestedForItemId** are scenario variables, whose value is set as -.
newRenewLoanRequestedForUserId =2
newRenewLoanRequestedForItemId=2
As Scenario grammar execution proceeds, an Observability **FindLoan**(2, 2) is executed which returns corresponding loanId, the return value is stored in scenario variable loanId.
loanId = 101.
Scenario grammar execution proceeds and a choice statement **"Choice(loanId > 0)true"** is evaluated . Scenario grammar execution enters the choice block. An Observability **GetLoanRenewalCount(101)** is executed . Method bound to ***GetLoanRenewalCount(101)*** would return the number of times the loan with Loan Id 101 has been renewed. ***GetLoanRenewalCount(101)*** returns **0**. Return value of Observability is stored in a scenario variable loanRenewalCount.
loanRenewalCount = 0.
Another choice statement "**Choice(loanRenewalCount < Parameters.MaxLoanRenewalCount) true"** is evaluated **Parameters.MaxLoanRenewalCount** had been set to 3 at the time of ACL-IUT binding. So Grammar execution proceeds inside the choice block, where another choice statement **"choice(UserHasPrivilege(newRenewLoanRequestedForUserId))true"** is evaluated. **UserHasPrivilege(2))** returns true. So the grammar inside choice block executes. Scenario grammar for instance **RENL1**, now waits to observe an event **loanRenewed(2,2**).

IUT should verify if the loan for the given userId and itemId can be renewed and call a method which would extend the loan. This method would take userId and itemId as parameters and return the positive LoanId if the loan renewal is successful, otherwise, it would return 0. Responsibility renewLoan(Integer userId, Integer itemId is bound to this method and will execute as -

renewLoan(2,2)

After all the post conditions have been evaluated in this responsibility, an Event loanRenewed(userId, itemId) is fired . userId, itemId are parameters for this responsibility so the event fired is loanRenewed(2, 2). Let's call it **Event51**.

Event51 - loanRenewed(2, 2)

Scenario grammar for instance **RENL1**, was waiting to observe the event **loanRenewed(2,2)**. **Event51** satisfies the scenario grammar and execution for instance **RENL1** continues further. An event **loanRenewedCompleted(2,2)** is fired , let's call this event **Event52**.

Event52 - loanRenewedCompleted(2,2)

Scenario instance **LLT2** was waiting to observe one of the three events - **loanExpired(2,2) loanRenewedCompleted(2,2) loanDeletionCompleted(2,2).** It observes **loanRenewedCompleted(2,2)** and continues its execution further. An Observability **FindLoan(2,2)** is used, which returns loanId as **101**, this return value is stored in scenario variable fineLoanId.

fineLoanId = 101

Next the scenario grammar for instance **LLT2** evaluates a choice statement **"choice(fineLoanId >0) true"**. Scenario grammar execution enters the choice block , where one more choice statement "**Choice(IsLoanOverdue(fineLoanId))" true** is evaluated. **IsLoanOverdue(101)** returns **false** hence the choice block gets skipped. Next, a redo statement causes the grammar execution to move to the outer choice block again. Which evaluates the choice statement **choice(fineLoanId >0) true**. again and execution goes inside the choice block - scenario grammar for instance **LLT2** again waits to observe one of the three events - **loanExpired(2,2) loanRenewedCompleted(2,2) loanDeletionCompleted(2,2).**

Scenario grammar for instance **RLBL1** was also waiting to observe an event **loanRenewedCompleted (2,2). Event56** satisfies the scenario grammar and execution for instance **RLBL1** continues further. Scenario grammar execution exits the choice block.

Terminate() statement executes and scenario instance **RLBL1** is destroyed.

Scenario grammar for instance **RENL1** continues its execution and exits the choice block. Contract level variable renewLoanRequestedForUserId, renewLoanRequestedForItemId are assigned 0 value, to signify that current request for loan renewal has been processed.

Terminate() statement executes and scenario instance **RENL1** is destroyed.

20- User u1 returns c2 of t2 First Return

Librarian requests return loan and enters the user and item information through library terminal. The method bound to responsibility **initiateReturnLoan(Integer userId, Integer itemId)** handles the input provided. Responsibility initiateReturnLoan executes as -

initiateReturnLoan(1, 5)

for this responsibility the value of parameters (userId, itemId) are stored in contract variable **returnLoanRequestedForUserId** and **returnLoanRequestedForItemId**.

returnLoanRequestedForUserId=1

returnLoanRequestedForItemId=5

After responsibility's post conditions are evaluated, a parameterized event **returnLoanRequested(returnLoanRequestedForUserId,returnLoanRequestedForItemId)** is fired. Let's call this event as **Event53**

Event53 returnLoanRequested(1,5)

Scenario **returnLoanCopy** is triggered on observing the event **returnLoanRequested(1,5) and** a new scenario instance is created, let's call this instance - **RL1**

newReturnLoanRequestedForUserId and **newReturnLoanRequestedForItemId** are scenario variables whose value is set as -

newReturnLoanRequestedForUserId =1

newReturnLoanRequestedForItemId =5

As Scenario grammar execution proceeds. an Observability FindLoan(newReturnLoanRequestedForUserId,newReturnLoanRequestedForItemId) is executed and the returned value is stored in scenario variable loanId. **FindLoan(1,5)** returns an integer value **102**.

loanId = 102

Scenario grammar execution for instance **RL1** proceeds and a choice statement "**Choice(loanId > 0)true"** is evaluated.

Grammar execution enters the choice block, where another choice statement **"choice(IsLoanOverdue(loanId)) true"** is evaluated

Observability **IsLoanOverdue(loanId)** is executed as **IsLoanOverdue(102)** which return false. So the grammar execution will skip the choice block.

Another choice statement **"Choice(DoesUserHaveLateFee(newReturnLoanRequestedForUserId)) true"** is evaluated. **DoesUserHaveLateFee(1)** returns false. So the grammar execution will skip this choice block as well.

Scenario grammar execution for instance **RL1** now expects parameterized event **loanDeleted(1,5)** to be observed.

Within IUT, it should be verified that user is eligible to return the item, if so then a method for actually deleting the loan record should be called, which would have two parameters userId, ItemId, which are 1, 5 in this case. This method would return the loanId of the loan just deleted, if loan cannot be returned then it would return integer 0.

This method is bound to a responsibility **returnLoan(String userId, String itemId)** , which executes as

returnLoan(1, 5)

Within this responsibility, after all the post conditions have been verified, an event loanDeleted(userId, itemId) is fired. Let's call this event **Event54**

Event54- loanDeleted(1, 5)

Scenario instance **RL1** was waiting to observe this event loanDeleted(**1,5**). This event is observed and scenario grammar is satisfied and execution continues. A parameterized event

loanDeletionCompleted(newReturnLoanRequestedForUserId,newReturnLoanRequestedForItemId)) is fired. Let's call this event as **Event55**

Event55- loanDeletionCompleted (1,5)

Scenario grammar for instance **LLT3** was waiting to observe one of the three events **loanExpired(1,5)) loanRenewedCompleted(1,5) loanDeletionCompleted(1,5) .** it observes **loanDeletionCompleted (1,5)** and scenario grammar execution continues.

An Observability **FindLoan(1,5)** is executed, which returns loanId as **0**, because this loan has been deleted , this return value is stored in scenario variable fineLoanId.

fineLoanId = 0

Next the scenario grammar for instance **LLT3** evaluates a choice statement **"choice(fineLoanId >0) true",** the choice block is skipped. A redo statement causes the execution control to go back to the outer choice block **choice(fineLoanId >0) true,** since fineLoanId is 0, the entire choice block is skipped this time . A terminate() statement is executed now, which causes this instance **LLT3** to be destroyed.

Grammar execution for scenario instance **RL1** continues , a parameterized event **itemReturned(newReturnLoanRequestedForItemId)** is fired. Let's call this event as **Event56**

Event56- itemReturned(5)

Scenario instance **IL5**, was waiting to observe this event **itemReturned(5)**. This event is observed and scenario grammar for instance **IL5** is satisfied. Execution for instance **IL5** proceeds. Execution exits choice blocks. A **redo** is encountered, which is for choice statement **Choice(DoesItemExist(aItem)) true**. Redo statement causes the execution to go back to the related choice statement. **DoesItemExist(5)** returns true. Grammar execution enters the choice block, where another choice statement "**Choice(IsItemAvailable(aItem)) true"** is evaluated. **IsItemAvailable(5)** returns true. Grammar execution enters the choice block,

now Scenario instance **IL5** waits to observe one of the two event **itemBorrowed(5)** or **itemDeletionCompleted(5)** to happen.

Scenario grammar for instance **RL1** exits the choice block. Contract variables ***returnLoanRequestedForUserId***, ***returnLoanRequestedForItemId*** are assigned value 0, to signify that request processing is complete for current return loan request. Terminate statement executes next and the scenario instance **RL1** is destroyed.