Reusing Transaction Models for Dependable Cloud Computing

Barbara Gallina
Mälardalen Real-Time Research Centre,
Department of Computer Engineering,
Mälardalen University,
Västerås, Sweden

Nicolas Guelfi
Laboratory for Advanced Software Systems
University of Luxembourg
6, rue R. Coudenhove-Kalergi,
L-1359 Luxembourg

ABSTRACT

Cloud computing represents a technological change in computing. Despite the technological change, however, the quality of the computation, in particular its dependability, keeps on being a fundamental requirement.

To ensure dependability, more specifically reliability, transaction models represent an effective means. In the literature, several transaction models exist. Choosing (reusing entirely) or introducing (reusing partially) transaction models for cloud computing is not an easy task. The difficulty of this task is due to the fact that it requires a deep understanding of the properties that characterize transaction models to be able to discriminate reusable from non reusable properties with respect to cloud computing characteristics. To ease this task, the PRISMA process is introduced. PRISMA is a Process for Requirements Identification, Specification and Machine-supported Analysis that targets transaction models. PRISMA is then applied to engineer reusable requirements suitable for the achievement of the adequate transaction models for cloud computing.

INTRODUCTION

Cloud computing is a computing service offered over the Internet. That is, a customer plugs into the "cloud" (metaphor for Internet) and uses computing scalable capabilities owned and operated by the service provider (Bernstein & Newcomer, 2009).

Two kinds of service can be offered over the Internet: either an application-specific service that offers a specific application (e-mail, search, social networking, etc.) or a general-purpose service (raw storage, raw processing power, etc.). The provider of these kinds of service may be a large company that owns many data centers, clusters of hundreds of thousands of computers.

Cloud computing represents a paradigm shift, a technological change in computing. This technological change forces cascading changes. Despite the technological change, however, the quality of the computation, in particular its dependability, keeps on being a fundamental requirement.

To ensure dependability, more specifically reliability, transactional principles, in particular ACID (Atomicity, Consistency, Isolation and Durability) properties (Härder & Reuter, 1983), which characterize the flat transaction model (Gray, A Transaction Model, 1980), represent an effective means. ACID properties combine fault tolerance and concurrency control to preserve global data consistency.

In the context of cloud computing, however, ACID properties are too strict and need to be reviewed and carefully changed, "relaxed", to achieve adequate transaction models, characterized by the right ACIDity (the right choice in terms of Atomicity, Consistency, Isolation and Durability). The traditional Atomicity, for instance, has to be relaxed when a computation is executed over a series of Internet's partitions that belong to different and autonomous service providers. Autonomy implies the possibility to decide locally about the outcome of the computation. This possibility would be denied by the *all-or-nothing semantics* (known as Failure Atomicity) that characterizes traditional atomicity, since it subordinates local decisions to the global decision. To preserve autonomy, the *all-or-nothing* semantics has to be changed into *all-or-compensation* semantics (known as compensation (Levy, Korth, & Silberschatz, 1991)).

Scattered throughout the literature are available several relaxed notions of the traditional ACID properties. Since, however, no effective means exist to support a systematic understanding of the differences and similarities among these notions, no selectable and composable on-the-shelf-(relaxed)-ACID properties exist yet. This lack of means hinders the beneficial exploitation (reuse) of these properties.

To reduce time to market and increase quality, reuse has to be the key-leading-principle and changes have to be introduced only where needed. Changes have to be engineered. To be able to plan the changes as well as the reuse correctly and efficiently, a methodological support has to be provided. In particular, the methodological support should help engineers to identify what has to be changed and what has to be kept unchanged. More specifically, it is fundamental to be able to identify what are the changes in terms of ACIDity, that is what are the changes required to adapt each single ACID property to meet the requirements of cloud computing and provide the right transaction model. By being able to identify what has changed and what remains unchanged, engineers are able to maximize reuse.

As initially discussed in (Gallina & Guelfi, A Product Line Perspective for Quality Reuse of Development Framework for Distributed Transactional Applications, 2008), to succeed in engineering systematically common (what remains unchanged) and variable properties (what changes), a product line perspective on transaction models has to be considered.

This chapter builds on this initial discussion and presents PRISMA (Gallina, PRISMA: a Software Product Line-oriented Process for the Requirements Engineering of Flexible Transaction Models, 2010). PRISMA integrates a product line perspective and supports the reuse of reliability-oriented and transaction-based requirements for achieving the adequate ACIDity. PRISMA is an acrostic that stands for Process for Requirements Identification, Specification and Machine-supported Analysis. PRISMA is helpful as a prism (from Greek "*prîsma*") in the identification of fundamental constituting properties of transaction models to achieve, as a result of the PRISMA process, correct and valid requirements specifications.

By integrating a product line perspective, PRISMA allows similarities and differences, which are called commonalities and variabilities in the terminological framework of product lines, to be identified, systematically organized and engineered to distinguish, as well as to derive, the single "products". Specifically, PRISMA is conceived for engineering the specification of a transaction model by placing the effort in revealing its requirements in terms of ACIDity. By following the PRISMA process, then, the adequate transaction models for cloud computing can be obtained.

PRISMA proposes two phases: the first one to engineer the commonalities and the variabilities that characterize the entire product line and the second one to derive the products by reusing the commonalities and the variabilities engineered during the first phase. The ACIDity, for instance, is seen as an abstract variability which can be customized during the application engineering phase to obtain the desired ACIDity of the transaction models (products). This customization is obtained by selecting and composing the adequate notions of ACID properties.

Each PRISMA's phase consists of three activities: identification, specification and verification & validation. To perform the activities, PRISMA proposes to use the requirements elicitation template, called DRET (Gallina & Guelfi, A Template for Requirement Elicitation of Dependable Product Lines, 2007), and the specification language, called SPLACID (Gallina & Guelfi, SPLACID: An SPL-oriented, ACTA-based, Language for Reusing (Varying) ACID Properties, 2008; Gallina, Guelfi, & Kelsen, Towards an Alloy Formal Model for Flexible Advanced Transactional Model Development, 2009).

DRET allows requirements engineers to gather requirements in a structured way. Domain concepts as well as product behaviors may be elicited through, respectively, DOMET and UCET, which are the two templates composing DRET.

SPLACID is a domain-specific specification language targeting transaction models. SPLACID integrates the above mentioned product line perspective and provides constructs for the specification of commonalities and variabilities within the product line. SPLACID offers a powerful means to maximize reusability and flexibility.

The SPLACID language benefits from a formal tool-supported semantics, which is obtained as a translation of the SPLACID concepts into Alloy concepts (Jackson, 2006). The Alloy-Analyzer tool (Alloy Analyzer 4), therefore, can be exploited to carry out automatic analysis. Because of its tool-supported semantics SPLACID contributes to improving the verifiability and reliability of transaction models.

In addition to the introduction of the PRISMA process, this chapter illustrates its application. PRISMA is applied to show how it can be used to engineer reusable Atomicity-related assets suitable for cloud computing.

The illustration of the PRISMA process is based on the current and informal understanding concerning the feasible ACIDity in the context of cloud computing (Karlapalem, Vidyasankar, & Krishna, 2010; Hohpe, 2009; Puimedon, 2009; Vogels, 2008; Pritchett, 2008). The aim of the chapter is to present a methodological support, the PRISMA process, and provide a first analysis on how it could be applied to reuse reliability-oriented and transaction-based models in the context of cloud computing. Cloud computing is still a rather new technology and therefore the PRISMA phases will need multiple iterations before achieving the right ACIDity.

BACKGROUND

This section is devoted to presenting the background concerning the problem and the solution space. First of all, this section introduces the main characteristics of cloud computing that make ACID properties inadequate to meet the dependability's requirements. Then a step-by-step immersion into transactional principles is given. This immersion is aimed at providing the necessary elements to understand the problem space and to prepare the reader to the solution space. In particular, first, the traditional transactional properties, namely ACID properties, which allow global data consistency to be preserved, are recalled. More specifically, the role of each single property is pointed out to deeply understand the property's contribution in preserving global data consistency. Then, intuitions are given to motivate and start conceiving the potentially numerous relaxed notions of the ACID properties as well as the potentially numerous transaction models that incorporate them. Two existing transaction models are then analyzed.

Finally, before letting the main section to introduce the solution proposal, the background section provides the fundamental ingredient of the solution space: product line engineering.

Cloud computing characteristics

As mentioned in the introduction, in cloud computing, the customer plugs into the cloud to use the computing service that is offered over it. The cloud is a metaphor for the Internet, a medium that consists of geographically and purposely scattered computers or supercomputers that perform different parts of the computation. Several properties characterize cloud computing (Buyya, Yeo, Venugopal, Broberg, & Brandic, 2009), the following list focuses on those that make the traditional ACID properties inadequate to meet the dependability's requirements of cloud computations.

- **autonomy**. The scattered computers and supercomputers that compose the cloud are loosely coupled and they belong to different autonomous organizations. Since autonomy implies the possibility to take decisions locally, it is not compatible with any master-slave hierarchy in which a local decision taken by a slave is subordinated to the decision of a non-local master.
- **complexity**. The computations involved in cloud computing are often complex. Complex comes from Latin past participle "complexus" and it means "composed of two or more parts". Complex computations are constituted of several operations accessing several data (complex data) and they often present parallelism. E-scientific applications, for instance, often submit to the cloud the request of executing large computations (i.e. large-matrix multiplications).
- **intra and cross-organization cooperation**. The scattered computers or supercomputers act in concert to execute very large computations. Cooperation implies that the scattered computers exchange information.
- **performance**. The computations involved in cloud computing are often expected to be executed quickly.
- **customization**. Cloud computations are supposed to be customizable. The customization may, for instance, involve reliability.
- **scalability**. Expansion, as well as contraction, of the capabilities (i.e. storage, database, etc.) needed by cloud computations is expected. Scalability can be guaranteed by scaling either vertically or horizontally (Pritchett, 2008). Vertical scaling (or scaling up) consists in moving the application that needs to be scalable to larger computers. Horizontal scaling (or scaling out) consists in adding more computers (generally low cost commodities) to the system that runs the application. Since vertical scaling is expensive and limited to the capacity of the largest computer, in cloud computing, scaling is mainly achieved horizontally. Cloud computing is in fact supposed to guarantee "infinite" scaling (a seemingly inexhaustible set of capabilities).

  When an application is executed on a cluster of computers (on a scaled out system) some adjustments are required. For instance, its data have to be properly distributed (scattered and replicated to guarantee low-latency access). To do that, a functionality-based approach can be followed. Functionality-based scaling consists in creating groups of data by dividing data according to functionalities and then spreading as well as replicating the groups across databases.

  Horizontal scaling, however, suffers from network partitions. Since network partitions happen, they need to be tolerated.

ACID properties

Atomicity, Consistency, Isolation and Durability, widely known under the acronym ACID (Härder & Reuter, 1983), are four properties, which, if satisfied together, ensure high dependability and, more specifically, reliability. These properties combine fault tolerance and concurrency control. The definitions of these properties, adapted from (Gray & Reuter, Transactions Processing: Concepts and Techniques, 1993) are given in what follows. The definitions make use of some terms defined in the appendix. The terms are written in *italics*.

Atomicity: a *work-unit*'s changes to the *state* are atomic: either all happen or none happen (all-or-nothing semantics, known as failure atomicity). Atomicity guarantees that in case of failure, intermediate/incomplete work is undone bringing the state back to its initial consistent value.

Consistency: a work-unit is a correct transformation of the state. All the a priori constraints on the input state must not be violated by the work-unit (intra-work-unit, local, consistency).

Isolation: a set of work-units either is executed sequentially (no interference) or is executed following a serializability-based criterion (controlled interference).

Durability: once a work-unit completes successfully, its changes to the state are permanent.

All of these four properties aim at preserving a consistent state (global data consistency), that is the state that satisfies all the predicates on *objects*. To become familiar with these definitions and to really achieve a deep understanding of their impact, a simple example (partially inspired by (Besancenot, Cart, Ferrié, Guerraoui, Pucheral, & Traverson, 1997)) is introduced to illustrate them. Throughout the example the following notation is used:

-to refer to a *read operation* which belongs to a work-unit labelled with the number 1 and which reads an object x (where the read value is v), the following notation is used: read1[x, v];

-similarly, to refer to a *write operation* which writes an object x (where the written value is v), the following notation is used: write1[x, v];

-the symbol "<>" denotes inequality and the symbol "*" denotes multiplication.

Example:

Two objects x and y of type integer are related by the constraint: y=2x

The initial state of the two objects is:

x=1 and y=2

Since 2 = 2 * 1, in the initial state, the constraint holds.

Case 1: ACID properties hold

Work-unit 1 executes permanently to completion (all semantics) and in isolation, the following operations: write1[x, 10] and write1[y, 20].

Work-unit 2 executes permanently to completion (all semantics) and in isolation, the following operations: write2[x, 30] and write2[y, 60]

A possible sequential execution:

write1[x, 10] write1[y, 20] write2[x, 30] write2[y, 60]

The final state in permanent storage is:

x=30 and y=60

Since 60= 2 * 30, the a priori constraint holds. The sequential execution of Work-unit 1 and Work-unit 2 therefore transform the state correctly and preserve the consistent state.

Case 2: AID properties only hold

Work-unit 1 executes permanently to completion (all semantics) and in isolation, the following operations: write1[x, 10] and write1[y, 30]

The final state in permanent storage is: x=10 and y=30

Since 30 <> 2*10, the a priori constraint does not hold. Work-unit 1 therefore does not transform the state correctly, i.e., it does not preserve the consistent state (**broken consistency semantics within the work-unit**).

Case 3: CID properties only hold

Work-unit 2 executes permanently in isolation but not atomically (something in the middle semantics) the following operations: write2[x, 10] and write2[y, 20]

The something in the middle semantics has to be intended as follows: only a subset of the operations to be executed is in reality executed. Work-unit 2, instead of executing both operations, executes only the first (write2[x, 10]).

The final state in permanent storage is:

x=10 and y=2

Since 2 <> 2 * 10, the a priori constraint does not hold. Work-unit 2 therefore does not preserve the consistent state (**broken all or nothing semantics**).

Case 4: ACI properties only hold

Work-unit 2 executes to completion (all semantics) and in isolation but not permanently the following operations: write2[x, 10] and write2[y, 20]. In particular, the second write operation.

The final state in volatile storage is:

x=10 and y=20

Since 20 = 2 * 10, the final state in volatile (non permanent) storage is consistent.

The final state in permanent storage is:
x=10 and y=2
Since 2 <> 2 *10, the a priori constraint does not hold. Work-unit 1 therefore does not preserve the consistent state (**broken durability semantics**).

Case 5: ACD properties only hold

Work-unit 1 executes permanently to completion (all semantics) but not in isolation the following operations: write1[x, 10] and write1[y, 20]

Its execution time overlaps the execution time of work-unit 2. Work-unit 2 executes permanently to completion but not in isolation the following operations: write2[x, 30] and write2[y, 60]

In particular considering the following interleaved execution:
write1[x, 10] write2[x, 30] write2[y, 60] write1[y, 20]
The final state in permanent storage is:
x=30 and y=20
Since 20 <> 2*30, the a priori constraint does not hold. The concurrent execution of Work-unit 1 and Work-unit 2 therefore does not preserve the consistent state (**broken isolation semantics, in particular an update is lost**).

ACID properties are not easy to ensure. Some research has shown that to guarantee these properties, a work-unit has to exhibit additional properties. A non-exhaustive list of these additional properties includes:

- each work-unit presents a short execution time (Gray & Reuter, Transactions Processing: Concepts and Techniques, 1993; Besancenot, Cart, Ferrié, Guerraoui, Pucheral, & Traverson, 1997).
- each work-unit accesses, during its execution time, a small number of data (Gray & Reuter, Transactions Processing: Concepts and Techniques, 1993; Besancenot, Cart, Ferrié, Guerraoui, Pucheral, & Traverson, 1997);
- the same data must not be accessed by a large number of concurrent work-units (Gray & Reuter, Transactions Processing: Concepts and Techniques, 1993; Besancenot, Cart, Ferrié, Guerraoui, Pucheral, & Traverson, 1997);
- each work-unit accesses only non-structured (simple) data (Gray & Reuter, Transactions Processing: Concepts and Techniques, 1993; Besancenot, Cart, Ferrié, Guerraoui, Pucheral, & Traverson, 1997);
- each work-unit executes reversible work (Gray & Reuter, Transactions Processing: Concepts and Techniques, 1993; Besancenot, Cart, Ferrié, Guerraoui, Pucheral, & Traverson, 1997);
- each work-unit is executed in a non-mobile environment (mobile work-units are assimilated into long-living work-units because of long communication delays over wireless channels, whether or not disconnection occurs, that is whether or not communication connections are broken) (Walborn & Chrysanthis, 1995);
- each work-unit accesses only data belonging to a single organization (belonging to a trust boundary) (Webber & Little).

This partial list helps in defining the limits beyond which it's hard or even counterproductive to guarantee ACID properties. This list is constantly being enriched as a consequence of the continuously challenging ACID properties in new environments. As discussed in the following sub-section, as soon as a work-unit does not exhibit these properties, the ACID properties have to be relaxed.

Relaxed ACID properties

Relaxed ACID properties are the result of a modification of the semantics of ACID properties to achieve less restrictive properties. Further, the relaxation allows the properties to meet the new requirements imposed by the application domains, which are different from those for which the original semantics was adequate. One-by-one each one of the fundamental ACID properties has been challenged in various environments to achieve realistically applicable properties, even though less simple. As a result, for each property, a spectrum of notions is available. In the following discussion are given the reasons that may lead to relax the ACID properties.

Relaxed atomicity is introduced to deal with the higher abortion frequency of longer running work-units by providing a means for guaranteeing intermediate results and selective roll-back (degradation acceptance, i.e., something in the middle semantics instead of all or nothing). Relaxed atomicity is also introduced to deal with computations that involve autonomous work-units (Levy, Korth, & Silberschatz, 1991).

The notions of atomicity differ on the basis of the allowed intermediate results. In (Derks, Dehnert, Grefen, & Jonker, 2001) intermediate results are interpreted as partial execution of operations.

Relaxed consistency is introduced to deal with the complexity of highly distributed systems. In case of complex distributed systems, when global consistency is not achievable, a relaxed consistency, for instance, allows a state to associate to a name a value that violates its domain range (Drew & Pu, 1995; Sadeg & Saad-Bouzefrane, 2000).

The notions of consistency differ on the basis of the allowed integrity violation. The domain range for instance could be violated according to a planned delta and the delta used is a criterion to differentiate the notions.

Relaxed isolation is introduced to deal with: 1) performance requirements (Adya, Liskov, & ÓNeil, 2000; Berenson, Bernstein, Gray, Melton, O'Neil, & O'Neil, 1995); 2) the higher data unavailability of long running work-units; and 3) cooperative work-units (Ramamritham & Chrysanthis, 1996).

The notions of isolation differ on the basis of the interference allowed.

Relaxed durability is introduced to deal with time constraints. In case of time constraints, write operations on persistent storage (which represent bottlenecks) have to be delayed and, in case of failures, data are lost. Relaxed durability is also introduced when permanence is not required immediately after the completion of a work-unit (Moss, 1981).

The notions of durability differ on the basis of the allowed loss.

Transaction models

Transaction models represent a means to structure complex computations by grouping logically related operations into sets and by imposing a series of properties on them. During the last three decades, several transaction models have been proposed.

The flat transaction model, known as ACID transactions, was the first transaction model. This model, as it will be explained in the remaining part of this subsection, exhibits ACID properties. All the others models have been obtained from it by relaxing ACID properties in some way. Since these other models stem from the same model, they must have precise similarities and differences.

To try to achieve a deeper understanding of their similarities and differences, they have been deeply surveyed (Elmagarmid, 1992; Gray & Reuter, Transactions Processing: Concepts and Techniques, 1993). In (Elmagarmid, 1992), for instance, transaction models are classified by taking into consideration two important aspects: 1) the structure that they impose on a *history*; and 2) their difference with respect to ACIDity. This classification is an important starting point in revealing the dimensions according to which the original transaction model has evolved. However, since the classification assumes the final user's point of view, the dimensions do not present a satisfying level of granularity. This subsection builds on this embryonic classification and provides a more detailed analysis. Two transaction models are discussed: the Flat Transaction Model and the Nested Transaction Model. The discussion makes use of the terms defined in the appendix. The terms are written in *italics*.

The Flat Transaction model (Gray, A Transaction Model, 1980) identifies the first *transaction model*. This model presents a very peculiar multigraph as structure: the multigraph is constituted of a single node and no edges. All the work-units compliant to this model, therefore, are typed according to a single *transaction type*:

- **flat** transaction type. Work-units of type flat exhibit the following properties: Atomicity, Consistency, Isolation and Durability. Moreover, their events are delimited by a standard boundary, that is two management events mark the initiation and termination of the work-unit and all the other events are in between.

The Nested Transaction Model (Moss, 1981) represents a *transaction model* that allows work-units to be structured in a hierarchical way, forming either a tree or an entire forest. In this model, work-units are partitioned into two distinct *transaction types*:

- top-level transaction type, usually called **root**. A top-level transaction type is equal to the Flat transaction type.
- nested transaction type, usually called **child**. Work-units of type child exhibit the following properties: Atomicity, Consistency, Isolation and Conditional Durability. Conditional Durability requires that whenever a work-unit of type child successfully terminates its work, it is not allowed to save it permanently but it has to delegate the work-unit, enclosing it, to take care of the durability of the work). Moreover, the events of the work-units of type child are delimited by a non-standard boundary. In particular, in the non-standard boundary the durability management events follow the event that marks the termination of the work-unit.

These two types constitute the two nodes of the multigraph that identify the Nested Transaction Model. A structural dependency, more precisely a containment dependency (obtained as an initiation dependency plus a termination dependency), relates these two types. Another containment dependency relates the child type with itself (a loop).

This model differs from the Flat Transaction Model according the following dimensions: structure (multigraph) and properties (a relaxed durability is introduced).

Software product line engineering

Software product line engineering is a key ingredient to maximize reuse systematically. In software product line engineering, reuse embraces the entire software life-cycle. The maximization of reuse is achieved thanks to the identification/engineering of common (always reusable) and variable (not always reusable) properties that characterize a set of products. Given its effectiveness in maximizing reuse, product line engineering can be a key-ingredient to reuse transaction models in the context of cloud computing. This subsection therefore is aimed at introducing some basic concepts related to software product line engineering and at drawing the attention to the current practices for product line requirements engineering to learn fruitful lessons to engineer the requirements of the adequate transaction model for cloud computing by maximizing reuse.

Basic concepts

The definitions of concepts listed below are mainly taken from (Clements & Northrop, 2001, Withey, 1996; Klaus, Böckle, & van der Linden, 2005).

- An **asset** is a description of a partial solution (such as a component or design document) or knowledge (such as a requirements database) that engineers use to build or modify software products.
- A **software product line** is “a set of software intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way”.
- The common set of core assets is known as commonalities. The set of assets that distinguish one product from another is known as variability.
- A **commonality** is a property that is common to all members of a software product line.

- A **variability** represents the ability of an asset to be changed, customized, or configured for use in a particular context. To characterize variability in more detail, it is useful to answer the following questions: "what varies?" and "how does it vary?". The answers to these questions lead to the definitions of variability subject and variability object, which are given here:
    - A **variability subject** is a variable item of the real world or a variable property of such an item;
    - A **variability object** is a particular instance of a variability subject.
- A **variation point** is a representation of a variability subject within domain artifacts enriched by contextual information.
- A **variant** is a representation of a variability object.
- **Feature** is a product's property that is relevant to some stakeholder and is used to capture a commonality or discriminate among products of the product line.
- **Feature diagrams** are trees (or graphs) that are composed of nodes and directed edges. The tree root represents a feature that is progressively decomposed using mandatory, optional, alternative (exclusive- OR features) and OR-features. A mandatory feature represents a product line commonality. Features that have at least one direct variable sub-feature (i.e. as one of its children) represent variation points.
- **Cardinality-based feature diagrams** are feature diagrams that allow features to be annotated with cardinalities.
- A **domain** represents an area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area. This set of concepts and terminology corresponds to "set of core assets", which is mentioned in the definition of a software product line.
- **Domain engineering** is the first phase that has to be carried out during product line engineering. This phase is meant at defining and building the "common set of core assets" that serves as a base to develop the products. Domain engineering starts with a domain analysis phase that identifies/engineers commonalties and variabilities amongst the software product line members.
- **Application engineering**, known also as "product derivation", is the second phase that has to be carried out during product line engineering. This phase represents a complete process of constructing products from the domain assets. This phase covers the process (mentioned in the definition of a software product line) of developing a "set of software-intensive systems" from "a common set of core assets in a prescribed way".

Domain engineering and application engineering are intertwined, the former providing core assets that are "consumed" by the latter while building applications. As a result of the product derivation task, feedback regarding specific products can be acquired and used to improve the software product line core assets.

Product line requirements engineering

Product line requirements engineering embraces the engineering (elicitation, specification and verification & validation) of the requirements for the entire product line as well as for the single products. The set of requirements that concerns the entire product line contains all the common and variable requirements. The set of requirements that concerns a single product is derived from the set of requirements of the product line. This derived set, contains all the common requirements plus those requirements that allow the product to be distinguished from the other products (variability objects).

In the literature as surveyed in (Kuloor & Eberlein, 2002; Gallina, PRISMA: a Software Product Line-oriented Process for the Requirements Engineering of Flexible Transaction Models, 2010), several general-purpose processes for product line engineering are available. These processes embrace the entire software life-cycle and, since they are general-purpose processes, they do not suggest specific techniques for accomplishing all the process tasks. The only techniques that are suggested are those to be used during the requirements elicitation and specification. These techniques include cardinality-based feature diagrams and textual use-case scenario-based templates.

To perform the requirements specification, a common suggestion is also to use domain-specific languages. To perform verification & validation, no suggestion is given.

Despite the interest of these general-purpose processes, sharply focused processes are lacking.

WHICH TRANSACTION MODELS FOR CLOUD COMPUTING?

From the background section, it emerges that cloud computing represents a technological change and that this change is revealed by a series of key properties. To achieve dependable cloud computations, transactional principles represent an effective means. However, as it emerges clearly from the background, since cloud computations do not exhibit the typical properties that are needed to be able to ensure ACID properties, adequate transaction models for cloud computing need to be identified, either by reusing an existing relaxed transaction models or by reusing some key properties. To achieve adequate transaction models, it is fundamental to take into account what emerges from the background. In particular, it is fundamental to take into account that:

- to deal with the complexity of cloud computations, structured transaction models are needed. This consideration suggests that the adequate transaction models should be hierarchical and therefore composed by more than one transaction type (the multi-graph that characterizes the Nested transaction model could represent an interesting starting point);
- to deal with autonomous computations, an adequate relaxed atomicity is needed. This consideration suggests that the all-or-nothing semantics of the traditional Atomicity should, for instance, be replaced by all-or-compensation (Levy, Korth, & Silberschatz, 1991) or by failure-atomic-or-exceptional (Derks, Dehnert, Grefen, & Jonker, 2001);
- to deal with cooperative computations scattered across the cloud, an adequate relaxed isolation is needed. This consideration suggests that the serializability-based semantics of the traditional Isolation should, for instance, be replaced by cooperative-serializability (Ramamritham & Chrysanthis, 1996);
- to deal with computations that need to be executed at high levels of performance, an adequate relaxed durability, as well as an adequate relaxed isolation, is needed. This consideration suggests that:
  - the no-loss semantics of the traditional Durability should, for instance, be replaced by an ephemeral permanence similar to the one available in in-memory databases (Garbus, 2010);
  - the serializability-based semantics of the traditional Isolation should, for instance, be replaced by the PL1 Isolation (Adya, Liskov, & ÓNeil, 2000).

Besides these considerations, it is also relevant to point out that, since the scattered computers and supercomputers that compose the cloud are expected to share data, cloud computing is governed by the CAP theorem. This theorem, initially presented as a conjecture by Eric Brewer, during his Keynote talk at PODC (Brewer, 2000), and later proved in (Gilbert & Lynch, 2002), states that of three properties of shared-data systems (data consistency, system availability and tolerance to network partitions), only two can be achieved at any given time.

Despite the fact that all the three above-listed properties are desirable and expected, a trade-off is required. Before sketching the plausible trade-off in the framework of cloud computing, the definitions (Gilbert & Lynch, 2002) of these three properties are given.

Data consistency is defined as “any read operation that begins after a write operation completes must return that value or the result of a later write operation”. This definition refers only to a property of a single operation. Since the ACID properties guarantee global data consistency to be preserved, they are commonly associated to this definition. The data consistency achieved through the ACID properties, however, is a property that refers to a set of operations. This set of operations, thanks to the Atomicity property is treated as if it was a single operation. Therefore, the association that commonly is done between these two different notions of data consistency makes sense.

System availability is defined as “every request received by a non-failing node in the system must result in a response”.

Partition tolerance is defined as: “No set of failures less than total network failure is allowed to cause the system to respond incorrectly”.

As seen in the background, cloud computations have to be partition-tolerant. According to the CAP theorem, then a choice has to be made between availability and consistency.

In (Pritchett, 2008), the necessity of lowering the ACIDity of the ACID transaction model is advocated. The author suggests that it is time to break the unbreakable ACID paradigm and that it is time to recognize that consistency is not paramount to the success of an application. The author, then, uses the term BASE (Basically Available, Soft state, Eventually consistent) to denote a new paradigm. This term, initially introduced in (Brewer, 2000), taken from chemical reactions domain, underlines the necessity of having a less ACID transaction model. This less ACID transaction model must provide a lower guarantee in terms of global data consistency (i.e. eventual consistency). However, if some guarantees in terms of global consistency are desired, the process of relaxing the ACID transaction model by lowering its ACIDity must not lead to a neutral transaction model. To plan this relaxation process, a methodological support is needed.

From the background it also emerges that an important set of relaxed transaction models exists. Since, however, this set is not structured, it is not possible to understand, compare and reuse these models easily. As a consequence it is not easy to understand which relaxed transaction model could be adequate for cloud computing or which properties could be reused to introduce a new transaction model to satisfy specific cloud computing requirements.

To ease the understanding, comparison and reuse, the set of transaction models has to be systematically organized. To do that, the dimensions that contribute to their differences and similarities have to be systematically engineered.

As discussed in the background, the current practices adopted within the product lines’ community can represent a key- ingredient in providing a solution to maximize reuse.

To structure the set of transaction models in order to ease the understanding, comparison, reuse as well as the careful planning of the adequate ACIDity, a product line perspective is adopted.

This section presents the product line perspective on transaction models. Then, it presents a new process for engineering the requirements of the desired transaction model. This new process integrates the product-line perspective. Finally, this section illustrates the application of the approach towards the achievement of the adequate transaction model for cloud computing.

A product line perspective on transaction models

As it was pointed out previously, existing transaction models constitute an unstructured set and as a consequence it is difficult to understand, compare and reuse existing transaction models. Product line engineering represents a concrete opportunity to engineer commonalities and variabilities and, as consequence, to maximize reuse systematically.

This subsection builds on the work proposed in (Gallina & Guelfi, A Product Line Perspective for Quality Reuse of Development Framework for Distributed Transactional Applications, 2008) and enhances it by introducing a more mature product-line perspective on the set of transaction models.

The viability and feasibility of the product line composed of transaction models is justified by the fact that since transaction models have more properties/features in common (commonalities) than properties/features that distinguish them (variabilities), there is more to be gained by analyzing them collectively rather than separately.

From what was presented in the background, two main commonalities characterize transaction models:
1- A transaction model can be seen as a multi-graph. Transaction types identify the nodes; structural dependencies inter-relating transaction types identify the edges.
2- Each transaction model is characterized by a specific ACIDity. The ACIDity of a transaction model can be seen as the result of the selection and composition of the ACIDities characterizing the transaction types, which compose the transaction model itself. The ACIDity consists of the coexistence and synergy of four assets aimed at guaranteeing Atomicity, Consistency, Isolation and Durability (Gray & Reuter, Transactions Processing: Concepts and Techniques, 1993). All these assets, taken individually, identify a variability subject, which may give rise to different variability objects. As seen in the background, the semantics of the ACID properties has been challenged in various ways. As a consequence, for each property a spectrum of notions is available. The notions which compose a spectrum are the variability objects which instantiate the corresponding variability subject. Failure Atomicity, for instance, represents a variability object. To obtain the appropriate Atomicity asset, the adequate variability object has to be selected. A transaction model is, therefore, characterized by an ACIDity, which results from the selection and composition of the variability objects that correspond to the (relaxed) ACID notions.

Whenever an adequate variability object is not available, if reasonable, the product line should evolve and a new variability object should be introduced to cover a specific need.

Figure 1 and Figure 2 show the cardinality-based feature diagram that partially represents the product line constituted of transaction models. In this diagram, variability subjects are represented by variation points, that is by those features that have at least one direct variable sub-feature. Leafs, instead, are selectable variants at variation points and represent variability objects.

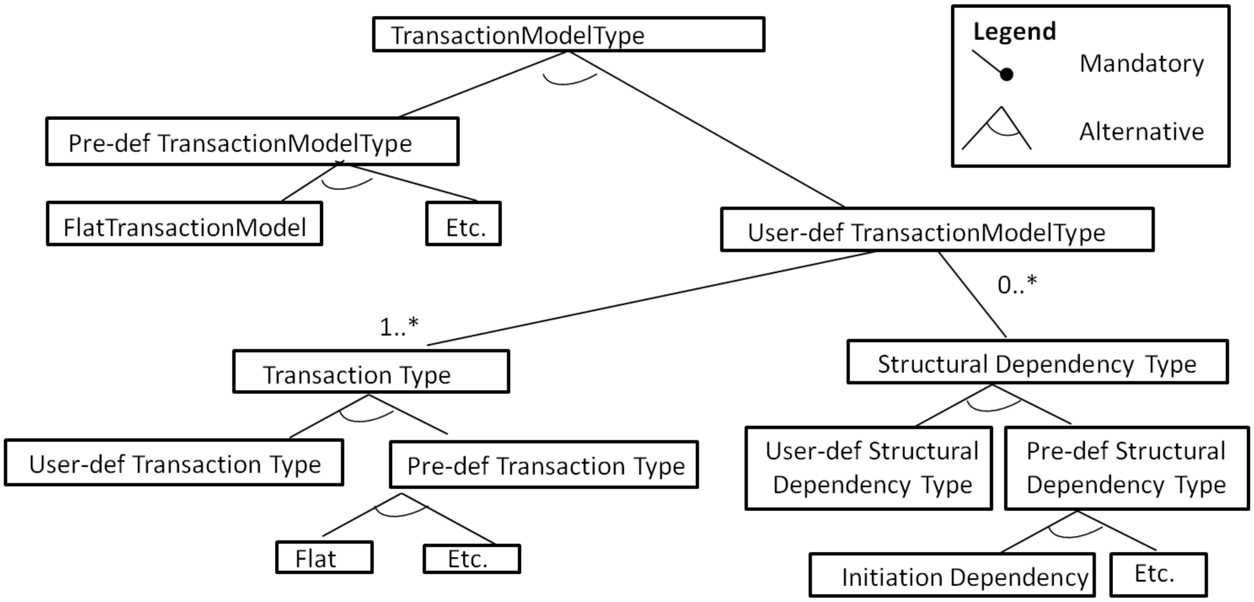

TransactionModelType
Legend
Mandatory
Alternative
Pre-def TransactionModelType
FlatTransactionModel
Etc.
User-def TransactionModelType
0..*
1..*
Transaction Type
Structural Dependency Type
User-def Transaction Type
Pre-def Transaction Type
User-def Structural Dependency Type
Pre-def Structural Dependency Type
Flat
Etc.
Initiation Dependency
Etc.

Figure 1 Cardinality-based feature diagram: focus on the multi-graph structure

Figure 1, in particular, focuses on the common multi-graph structure and it summarizes that a transaction model can be either a pre-defined transaction model (i.e. the Flat transaction model discussed in the background) or a user-defined one. A transaction model is commonly structured as a collection of inter-dependent transaction types. Transaction types and the structural dependencies that inter-relate them can in turn either be selected among those available (pre-defined) or obtained by selecting and composing the needed sub-features.

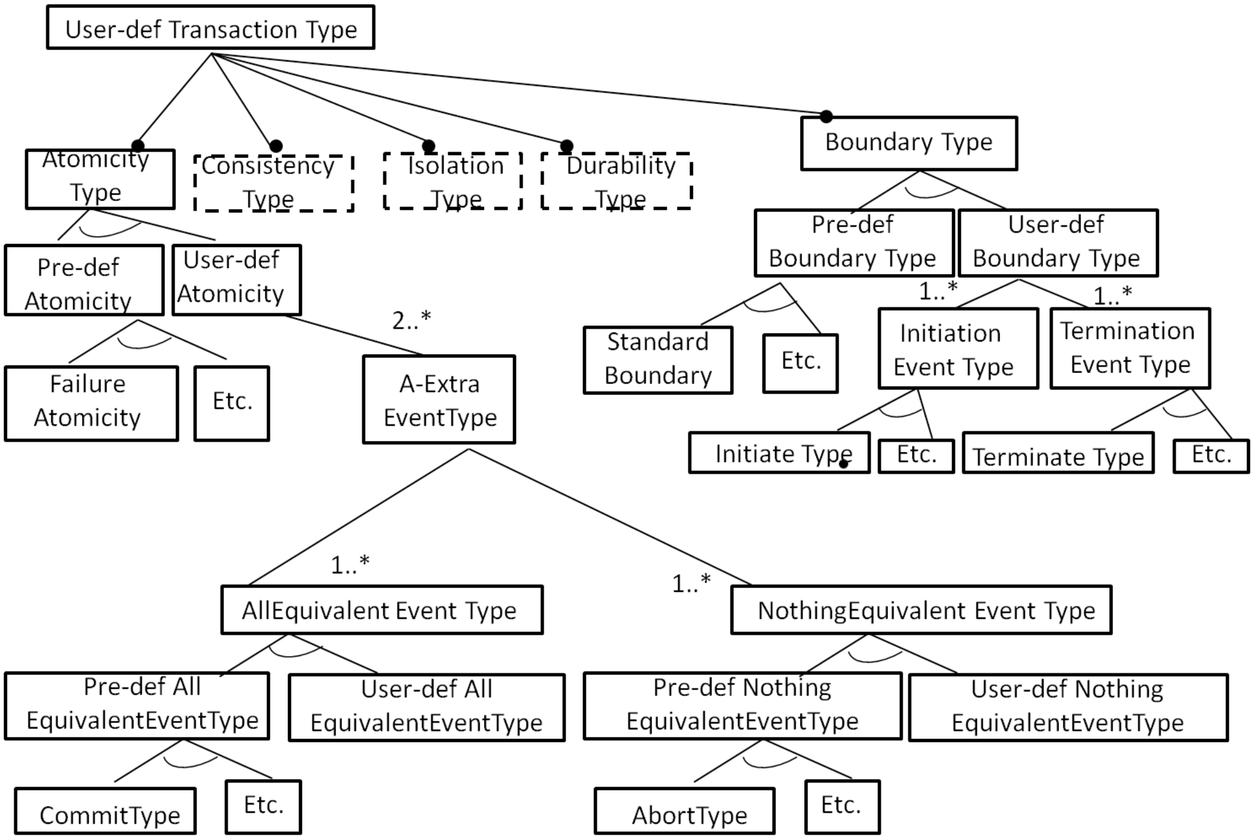


Figure 2 Cardinality-based feature diagram: focus on the transaction type's Atomicity

Figure 2 shows that each transaction type is characterized by commonalities (i.e. coexistence of a boundary, an Atomicity variant, a Consistency variant, an Isolation variant and a Durability variant) and variabilities represented by, for instance, the ACID variation points. The Atomicity Type is a variation point and at this point a choice has to be made. A predefined Atomicity (i.e. Failure Atomicity) or a user-defined Atomicity has to be selected. A user-defined Atomicity requires the introduction of new features to describe the desired intermediate result. Usually, to mark the successful or unsuccessful execution of an operation, specific management events are used. For instance, the all semantics is represented by marking all the operations with a management event (a Commit Type event) that indicates full commitment. Similarly, the nothing semantics is represented by marking all the operations with a management event (an Abort Type event) that indicates full abortion. Management events are helpful to distinguish the different intermediate semantics. In Figure 2, these events, which are used to distinguish the Atomicity variants, are represented by the feature called “A-Extra Event Type”.

For space reasons, dashed features are not detailed. For more details, the interested reader can refer to (Gallina, PRISMA: a Software Product Line-oriented Process for the Requirements Engineering of Flexible Transaction Models, 2010).

The PRISMA process

As mentioned in the background, within the product line's community, a sharply focused methodological support is lacking. As stated in (Jackson M., 1998), it's a good rule of thumb that the value of a method is inversely proportional to its generality. A good method addresses only the problems that fit into a particular problem frame. Systematic and sharply focused methods help in reaching a solution.

To reuse existing transaction models either entirely or partially a sharply focused methodological support is needed. PRISMA provides that support. PRISMA stands for Process for the Requirements Identification, Specification and Machine supported Analysis. PRISMA is a new software product line-oriented requirements engineering process, which aims at being useful as a prism in revealing clearly the properties composing the transaction models.

PRISMA is compatible with the general-purpose processes surveyed in (Kuloor & Eberlein, 2002). PRISMA inherits from them. PRISMA, for instance, similar to those processes, is composed of the two typical inter-related phases: the domain engineering phase and the application engineering phase. PRISMA, however, aims at offering a sharply focused method as opposed to one that is more general-purpose. PRISMA targets a precise class of problems and aims at offering specific guidelines and techniques to perform the tasks, which make up the process. In particular, within PRISMA the following techniques are integrated:

- a specific use-case-based template to carry out the elicitation, called DRET (Gallina & Guelfi, A Template for Requirement Elicitation of Dependable Product Lines, 2007);
- a domain-specific specification language to carry out the specification, called SPLACID (Gallina & Guelfi, SPLACID: An SPL-oriented, ACTA-based, Language for Reusing (Varying) ACID Properties, 2008; (Gallina, PRISMA: a Software Product Line-oriented Process for the Requirements Engineering of Flexible Transaction Models, 2010);
- a tool support (integration of the Alloy Analyzer tool) to carry out the verification and validation (Gallina, Guelfi, & Kelsen, Towards an Alloy Formal Model for Flexible Advanced Transactional Model Development, 2009; (Gallina, PRISMA: a Software Product Line-oriented Process for the Requirements Engineering of Flexible Transaction Models, 2010);

Before introducing the static as well as the dynamic structures of the PRISMA process, a brief explanation of these techniques is given.

The PRISMA's techniques

As mentioned before, the relevance of the PRISMA process is its sharp focus. This sharp focus is achieved through the integration of sharply focused techniques, which support engineers during the different tasks that compose the process.

To ease the understanding of the following sections, these techniques (DRET, SPLACID, and the tool-supported verification and validation) are briefly introduced.

DRET is a requirements elicitation template suitable for the elicitation of dependable software product lines. This template is composed of two parts: a DOMain Elicitation Template (DOMET) and a Use Case Elicitation Template (UCET).

The DOMET allows the concepts of the domain to be elicited. The DOMET is depicted using a tabular notation. The field meaning is briefly provided in the following. **Name** labels the concept via a unique identifier. **Var Type** underlines commonalities and variabilities in the software product line and is filled with one of the following keywords: *Mand* (mandatory concept); *Alt* (alternative concept); *Opt* (optional concept). **Description** is an informal explanation of the concept purpose. **Dependencies** exposes any kind of relationship with other concept(s). **Misconception & class(es)** is an informal explanation concerning the misunderstanding of the domain (fault). **Misconception consequence & class(es)** is an informal explanation concerning the consequence (failure), observable by the stakeholders, that the misunderstanding of the domain may entail. **Priority Level** represents different levels of priority on the basis of criticality and is filled with one of the following keywords: *High*, *Medium* or *Low*.

The two fields that involve the misconception are meant to elicit the *causality chain* existing among the dependability's threats (Avizienis, Laprie, Randell, & Landwehr, 2004).

The UCET allows the software product line members' behaviour to be elicited. The UCET is an extension of the popular textual use-case scenario-based template given by Cockburn. It allows requirements to be organized in a tabular form that has a series of fields. Besides the standard fields (use case name, goal, scope, level, trigger, the actors involved, pre-conditions, end-conditions, main scenario, alternatives of the main scenario) the UCET includes fields (mis-scenarios, recovery scenarios, etc.) to elicit non-functional behaviour. The UCET may be labelled as *Collaborative* or *Single*. These labels, jointly with the labels used to characterize the resources, provide a means to distinguish concurrency's types. The explanation of the detailed structure of the UCET is outside of this scope. The names of the template's fields, however, should be understandable. Details can be found in (Gallina & Guelfi, A Template for Requirement Elicitation of Dependable Product Lines, 2007).

SPLACID is a domain-specific language conceived to specify transaction models on the basis of their fundamental properties. SPLACID is based on ACTA (Chrysanthis, 1991), a unified framework that was introduced for the specification, synthesis and verification of transaction models from building blocks. The main advantages of SPLACID over ACTA are:

- a well-structured abstract syntax, given in terms of a meta-model;
- a well-structured textual concrete syntax, given in EBNF;
- a formal semantics given following a translational approach.

SPLACID integrates the product line perspective, which was discussed previously. The meta-model that defines its abstract syntax contains an abstract meta-class for each concept that was labelled as User-def in Figure 1 and in Figure 2 (i.e. TrasactionModelType, TransactionType, etc). Figure 3 illustrates a blow up the SPLACID meta-model.

Similarly, the syntactical rules, which are written in EBNF, that define the concrete syntax contain a non-terminal for each concept that was labelled as User-def.

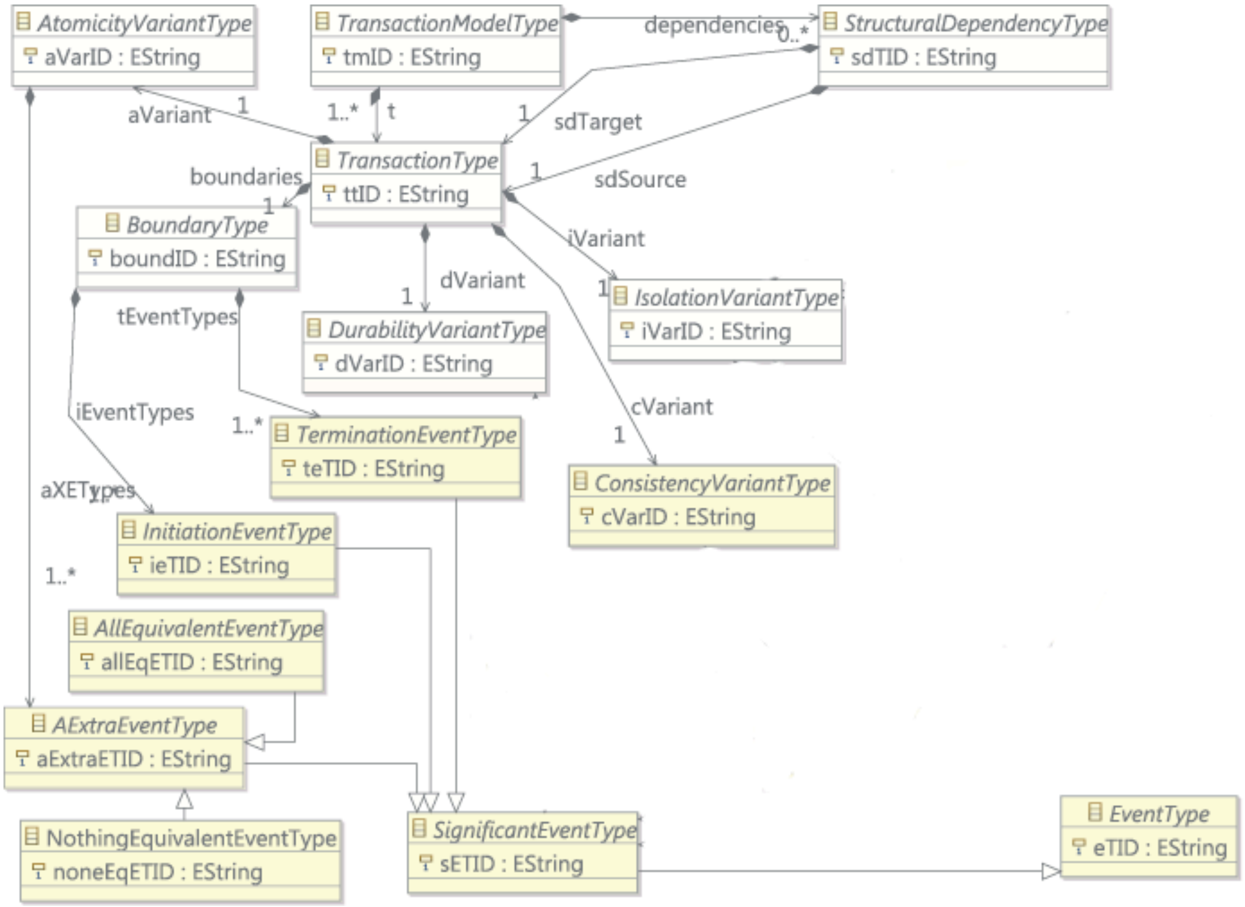


Figure 3 Cut of the SPLACID meta-model.

Each concept (variant) that was a leaf of a feature labelled Pre-def in Figure 1 and in Figure 2 is represented in the SPLACID meta-model by a concrete meta-class that reifies the corresponding abstract meta-class. The concept of Failure Atomicity, for instance, is represented by a concrete meta-class that reifies the abstract meta-class corresponding to the Atomicity Type.

Similarly, these leaf-concepts have a representation in the concrete syntax and specifically they are represented as terminals.

All the concepts (abstract and concrete) are characterized by a set of constraints that define them. The constraints impose either a specific order among the events that belong to a history or require specific management events to occur in the history.

The concept of Failure Atomicity, for instance, establishes that all events, which have executed an operation on an object, have to be followed by one management event. The management event in particular has to be of type CommitType in case the “all"’ semantics has to be guaranteed; it has to be of type AbortType in case the “nothing” semantics has to be guaranteed.

The tool support for verification and validation is achieved by providing a translational semantics to SPLACID. SPLACID concepts are translated into Alloy concepts. A model transformation provides rules to obtain an Alloy specification from a SPLACID specification. Once the Alloy specification is available, the tool, called Alloy Analyzer, can be used to verify and validate it. The verification consists in executing the executable Alloy specification to look for instances of it. If instances exist, it means that the specification is contradiction-free and that therefore it is consistent. A consistent specification has to be validated. To validate the specification, the instances are inspected to check that they really describe the desired requirements.

The PRISMA process' static structure

The backbone of the static structure is made up of a set of activities, a set of roles, and a set of work-products. A detailed description of these elements, using the SPEM 2.0 standardized format (OMG, 2008), is given in (Gallina, PRISMA: a Software Product Line-oriented Process for the Requirements Engineering of Flexible Transaction Models, 2010). In the following list, the set of activities is presented. For each activity, the roles, the work-products and the guidelines (if any) are given.

- **Product line requirements elicitation**. This activity is carried out by using the DRET template. This activity aims at revealing commonalities, variabilities (i.e. the Atomicity types, etc.) and dependencies (cross-cutting concerns) existing among variants at different variation points. The dependencies have to be documented to constrain the permitted combinations of variabilities.
  The work-products involved are:
  - input: none
  - output: the filled-in DRET template.

  The roles involved are:
  - Concurrency control expert focuses on the Isolation spectrum.
  - Fault tolerance expert (before termination) focuses on the Atomicity spectrum.
  - Fault tolerance expert (after termination) focuses on the Durability spectrum.
  - Application domains expert focuses on the Consistency spectrum.

  The first four roles represent key-roles in engineering the requirements pertaining to the ACID spectra.
  Guidelines: DRET usage in the context of the PRISMA process
- **Product line requirements specification**. This activity is carried out by using the SPLACID language. This activity aims at specifying the elicited requirements.
  The work-products involved are:
  - input: the filled-in DRET template
  - output: the SPLACID specification.

  The roles involved are:
  - Analyst mastering the SPLACID

  Guidelines: traceability rules to move from a filled-in DRET template to a SPLACID specification.
- **Product line requirements verification & validation**. This activity is carried out by using the Alloy Analyzer tool. This activity aims at verifying and validating the specified requirements. The work-products involved are:
  - input: the SPLACID specification
  - output: the Alloy specification

  The roles involved are:
  - Analyst mastering the Alloy Analyzer tool

  Guidelines: transformation rules to translate SPLACID concepts into Alloy concepts.
- **Product requirements derivation**. This activity is carried out by using the DRET template. This activity aims at eliciting the requirements of a single product. The elicitation is obtained by derivation/pruning (the thick black path in Figure 4), that is by selecting from the filled-in DRET template received in input all the commonalities plus those variability objects that characterize the product.
  The work-products involved are:

  - input: the filled-in DRET template
  - output: the pruned filled-in DRET template.

  The roles involved are:
  - Product requirements engineer. This role incorporates the competences of the experts in ACID spectra. In addition, this role is competent in pruning.
- **Product requirements specification**. This activity is carried out by using the SPLACID language. This activity aims at achieving the SPLACID specification of a single product.

  The work-products involved are:
  - input: product line SPLACID specification (the black dashed path in Figure 4 is followed) xor pruned filled-in DRET template (the thick and horizontal black path in Figure 4 is followed).
  - output: Product-SPLACID specification.

  The roles involved are:
  - Product requirements engineer.
  - Product specifier. This role incorporates the competence of Analyst mastering the SPLACID language. In addition, this role is competent in pruning.

  Guidelines: traceability rules to move from a filled-in DRET template to a SPLACID specification.
- **Product requirements verification & validation.** This activity is carried out by using the Alloy Analyzer tool. This activity aims at achieving a correct and valid Alloy specification of a single product.

  The work-products involved are:
  - input: product line Alloy specification (the thick grey path in Figure 4 is followed) xor product-SPLACID specification (the thick and horizontal black path in Figure 4 is followed).
  - output: Product-Alloy specification.

  The roles involved are:
  - Product verifier and validator. This role incorporates the competence of Analyst mastering the Alloy Analyzer tool. In addition, this role is competent in pruning.
  - Product user represents a key-role during validation.
  - Product requirements engineer

  Guidelines: transformation rules to translate SPLACID concepts into Alloy concepts.

The PRISMA process' dynamic structure

Besides the static structure, to fully define a process, its dynamic structure must also be provided. A software process defines ordered sets of activities, which may be grouped into phases. The ordered sets of activities define meaningful sequences that, if followed, allow interacting roles to produce valuable work-products. Figure 3, summarizes these meaningful sequences. Globally, Figure 3 shows that the PRISMA process is made up of two inter-related phases: the domain engineering phase and the application engineering phase.

During the domain engineering phase, commonalities and variabilities characterizing the product line are engineered. Artefacts, which constitute the product line's assets, are produced. Once the domain is rich enough to allow requirements engineers to derive at least one product from the assets available, the second phase can be started. During the application engineering phase, single products are derived.

The sequence of these two phases is iterated. The iteration ends as soon as the product line loses its worthiness. Four roles are in charge of establishing worthiness of iterating the sequence: the Product requirements engineer, Product specifier, Product verifier and validator, and Product user.

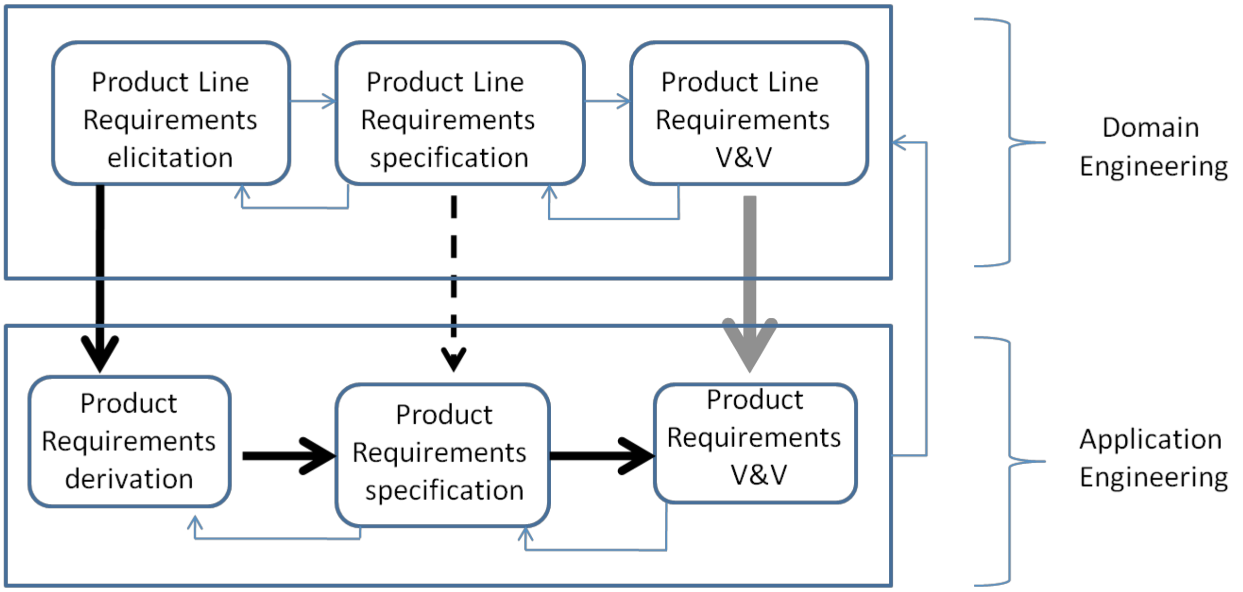

Product Line Requirements elicitation
Product Line Requirements specification
Product Line Requirements V&V
Product Requirements derivation
Product Requirements specification
Product Requirements V&V
Domain Engineering
Application Engineering

Figure 4 The PRISMA process' dynamic structure

Towards adequate transaction models for cloud computing

To achieve adequate transaction models to be used for cloud computing, the reusable assets have to be engineered. To do that the PRISMA process is applied. This subsection, first introduce a toy cloud computation to understand the characteristics that force a change in terms of ACIDity. Then on the basis of this toy computation, the PRISMA process is applied to achieve reusable Atomicity-related assets, adequate for cloud computing. In particular, the thick black path, shown in Figure 4, is followed until the achievement the SPLACID specification. The following ordered tasks are executed:

- Product line requirements elicitation;
- Product requirements derivation;
- Product line requirements specification.

The last step of the path, that is the Product requirements V&V, is not executed but simply discussed. The translation into Alloy is not presented since, to be fully understood, it requires wider background information, which for space reasons cannot be provided within this chapter.

Finally, this subsection discusses the results obtained by applying the PRISMA process.

Toy computation

The toy computation introduced here is adapted from (Pritchett, 2008). The computation involves the modification of three different objects which are related by a consistency constraint and which are distributed on different computers. The first object x contains the purchase, relating the seller and buyer and the amount of the purchase. The second object y contains the total amount sold and bought by a seller. The third object z contains the total amount bought by a buyer.

The modifications consist in one insert (a write operation) into the first object and in two updates (a read operation followed by a write operation) into the last two objects. The notation introduced in the background, has to be enriched to describe this computation. A read operation on an object x is denoted as read[x]. Therefore the computation consists of the following set of operations:

{write [x, 10]; read [y]; write [y, 10]; read [z]; write [z, 10]}

Due to the distribution of the objects, in (Pritchett, 2008), these modifications are not grouped altogether. Three different work-units are used to decompose the computation. These three work-units are:

Work-unit 1={write [x, 10]}
Work-unit 2={read [y]; write [y, 10]}
Work-unit 3={read [z]; write [z, 10]}

All these three work-units exhibit ACID properties. However, if at least one of them does not complete the work while the others do, the global consistency is not guaranteed. As it was explained in the background, global consistency would be broken due to the broken atomicity.

An additional work-unit, called for example Work-unit 4, should be considered to enclose these three modifications. This work-unit is:

Work-unit 4={write [x, 10]; read [y]; write [y, 10]; read [z]; write [z, 10]}

Work-unit 4 does not exhibit ACID properties. The Atomicity property, in particular, is replaced by a weaker notion of atomicity that allows a work to be done only partially.

Product line requirements elicitation

To plan the adequate Atomicity-related assets, first of all a deep understanding of the characteristics of the application domain is mandatory. This understanding must be documented. According to the PRISMA process, the *Application domains expert* has to fill in the DRET template with this information. Once this information is available, the elicitation of the ACIDity-related assets may start.

The *Fault tolerance (before termination) expert*, for instance, may proceed by filling-in the sub-DOMET and the UCET concerning Atomicity.

The sub-DOMET must contain all the concepts concerning the Atomicity-assets (all the Atomicity's notions and Atomicity's management events) and the UCET must contain all the behaviours.

The goal of the *Fault tolerance (before termination) expert* is to engineer the Atomicity-assets. With respect to Figure 2, the *Fault tolerance (before termination) expert* has to provide the User-def assets as summarized in Figure 5.

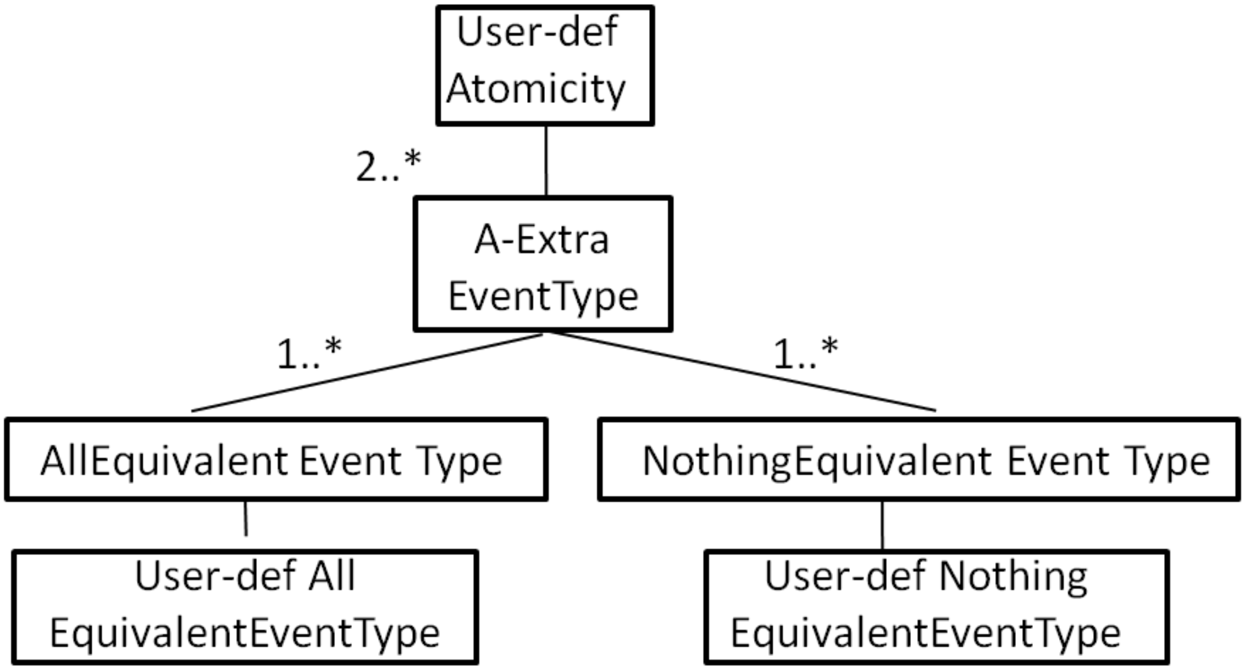
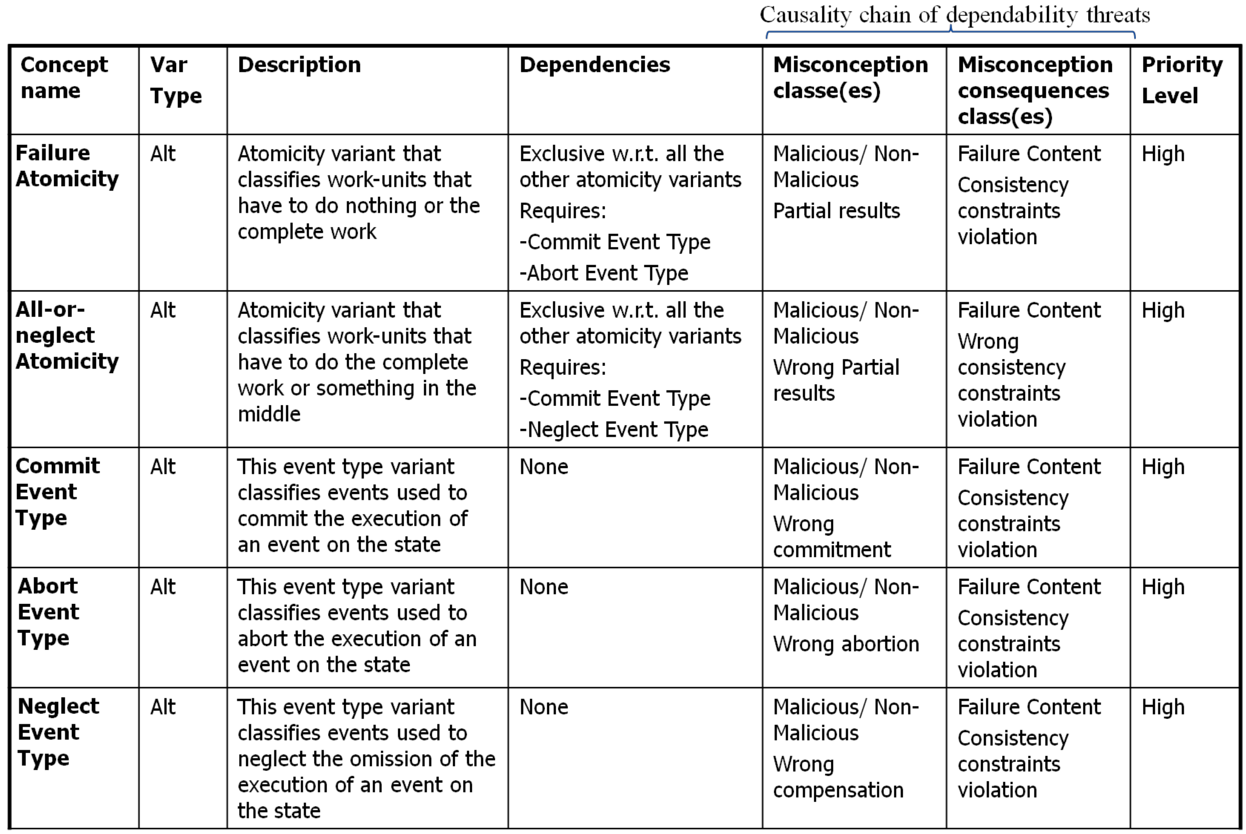

User-def Atomicity
2..*
A-Extra EventType
1..*
1..*
AllEquivalent Event Type
NothingEquivalent Event Type
User-def All EquivalentEventType
User-def Nothing EquivalentEventType

Figure 5 Atomicity-related assets

On the basis, of the toy computation, two notions of Atomicity seem to be necessary: the traditional notion of Atomicity (Failure Atomicity) and a new notion that allows the set of operations to complete fully or only partially. This new notion is called here “All-or-neglect Atomicity”.

As mentioned in the background, the notions of Atomicity differ from each other on the basis of the completeness of the execution. To distinguish these notions (these variants), different event types can be considered. For instance, the notion of Failure Atomicity (all or nothing semantics) might be characterized by two event types: one for the “all” semantics, aimed at committing the entire work (that is each event within the work-unit has to be committed), and the other for the “nothing” semantics, aimed at aborting the entire work (that is each event within the work-unit has to be aborted).

The sub-DOMET, presented in Figure 6, therefore, contains the two notions of Atomicity and the different notions of event types used to distinguish them.



Causality chain of dependability threats

Concept name	Var Type	Description	Dependencies	Misconception classe(es)	Misconception consequences class(es)	Priority Level
Failure Atomicity	Alt	Atomicity variant that classifies work-units that have to do nothing or the complete work	Exclusive w.r.t. all the other atomicity variants Requires: -Commit Event Type -Abort Event Type	Malicious/ Non-Malicious Partial results	Failure Content Consistency constraints violation	High
All-or-neglect Atomicity	Alt	Atomicity variant that classifies work-units that have to do the complete work or something in the middle	Exclusive w.r.t. all the other atomicity variants Requires: -Commit Event Type -Neglect Event Type	Malicious/ Non-Malicious Wrong Partial results	Failure Content Wrong consistency constraints violation	High
Commit Event Type	Alt	This event type variant classifies events used to commit the execution of an event on the state	None	Malicious/ Non-Malicious Wrong commitment	Failure Content Consistency constraints violation	High
Abort Event Type	Alt	This event type variant classifies events used to abort the execution of an event on the state	None	Malicious/ Non-Malicious Wrong abortion	Failure Content Consistency constraints violation	High
Neglect Event Type	Alt	This event type variant classifies events used to neglect the omission of the execution of an event on the state	None	Malicious/ Non-Malicious Wrong compensation	Failure Content Consistency constraints violation	High

Figure 6 Sub-DOMET focuses on Atomicity

A single UCET is used to elicit the requirements concerning the variants associated with the behaviour related to the guarantee of a property. The Single UCET filled in by the Fault tolerance (before termination) expert is:

- ID: UC2
- Single Use Case name: provide [V1] Atomicity
- Selection category: Mand
- Description: During this use-case, [V1] is guaranteed
- Primary Actor: Atomicity-Manager
- Resources: work-units (no sharing), data (competitive sharing), operations (no sharing)

- Dependency: if V1=Failure, then V3=aborts; if V1=All-or-neglect, then V3=neglects
- Preconditions: [V2] data
- Post-conditions: [V2] data
- Main scenario: The Atomicity Manager guarantees [V1] Atomicity, i.e. if the work done by the work-units is complete, it commits otherwise it [V3]
- Alternatives of the main scenario: None
- Variation points description:
  - V1: Type = Alt, values = {Failure, All-or-neglect}, Concerns = Behaviour
  - V2: Type = Alt, values = {Consistent, Eventually consistent}, Concerns = Behaviour
  - V3: Type = Alt, values = {aborts, neglects}, Concerns = Behaviour
- Non-functional: reliability
- Duration: None
- Location: None
- Mis-scenario: None
- Fault Variation descriptions: None
- Recovery scenarios: None

Product requirements derivation

The requirements of the single assets can be derived easily from the filled-in DRET previously obtained. Two different Atomicity assets can be derived. The derivation (pruning) of the concepts belonging to a single asset is obtained by:

- keeping all the mandatory concepts;
- choosing the desired concepts in the case of alternatives;
- choosing a concept or not in case of an option.

The resulting two pruned filled-in sub-DOMETs are:

- Pruned-filled-in-sub-DOMET related to Failure Atomicity={FailureAtomicity, CommitEventType, AbortEventType}
- Pruned-filled-in-sub-DOMET related to All-or-neglect Atomicity={All-or-neglectAtomicity, CommitEventType, NeglectEventType}

Despite the small dimension of this asset, by inspecting the two pruned sub-DOMETs, it can be seen that the set of common concepts is not empty.

Reused concepts ={CommitEventType}

The derivation (pruning) of the behaviour characterizing a single asset is obtained by:

- keeping all the mandatory UCETs;
- choosing the desired UCETs in case of alternatives, choosing a UCET or not in case of option;
- for each UCET derived for the product, selecting the desired variant at each variation point.

The two Atomicity assets differ on the basis of the choices that have to be made at the variation points. To obtain the behavior of the FailureAtomicity asset, in UC2 the following choices must be done:

- V1=Failure
- V2=consistent
- V3=aborts

To obtain the behavior of the FailureAtomicity asset, in UC2 the following choices must be done:

- V1=All-or-neglect
- V2=Eventually consistent
- V3=neglects

The two pruned UC2 differ only on the basis of the choices made at the variation points. The remaining part is in common and can be totally reused.

Product line requirements specification

The concepts and behaviours that have been previously documented must be specified using the SPLACID language. The SPLACID language offers extension mechanisms. The language is not closed but evolves in parallel with the evolution of the product line. As soon as new asset is introduced, the abstract syntax, the concrete syntax as well as the executable semantics have to evolve.

Therefore, the traceability rules detailed in (Gallina, PRISMA: a Software Product Line-oriented Process for the Requirements Engineering of Flexible Transaction Models, 2010) have to be followed to move from the work-product obtained during the elicitation (that is the filled in DRET template) to the SPLACID specification.

According to these rules, for each concept introduced in the DOMET, an abstract meta-class (if not yet available) and a concrete meta-class have to be added in the meta-model. Considering that the concepts related to Atomicity, introduced during the elicitation, represent variants of pre-existing concepts (the AtomicityVariantType, NothingEquivalentEventType, AllEquivalentEventType, etc. see Figure 5), only concrete meta-classes (namely, FailureAtomicity, All-or-neglectAtomicity, CommitType, etc.) have to be added. The concrete meta-classes represent reusable modeling concepts.

Similarly, terminals (namely, FailureAtomicity, All-or-neglectAtomicity, CommitType, etc.) have to be introduced in the concrete syntax.

The SPLACID specification for the Failure Atomicity is:

FailureAtomicity
AXE={CommitType, AbortType}

The SPLACID specification for the All-or-neglect Atomicity is:

All-or-neglect Atomicity
AXE={CommitType, NeglectType}

In the two-above SPLACID specifications, “AXE” stands for *Atomicity Extra Events* and it is the name of the set that contains all the management event types related to Atomicity.

Product line requirements V&V

The SPLACID concepts, newly introduced, have to be translated into the Alloy concepts. The translation rules detailed in (Gallina, PRISMA: a Software Product Line-oriented Process for the Requirements Engineering of Flexible Transaction Models, 2010) have to be followed. As a result of the translation an Alloy specification is available for both assets. As can be imagined, the two specifications differ on the basis of the concepts and behaviours that distinguish them. The remaining part is equivalent.

The resulting Alloy specification can be executed to check its consistency as well as its validity The Alloy Analyzer tool, for instance, is used to check that the All-or-neglect Atomicity allows a less constrained history.

With respect to the toy computation, more specifically with respect to Work-unit 4, the Alloy Analyzer tool, for instance, is expected to allow the instance *h* (given below) in the case of All-or-neglect Atomicity and forbid it in the case of Failure Atomicity.

h: write [x, 10] →read [y]→write [y, 10]

Discussion

Despite this brief and incomplete illustration of the PRISMA process, its effectiveness in easing the (partial or entire) correct and valid reuse of transaction models should be evident. The reuse embraces the entire set of the PRISMA’s work-products: elicited requirements (sub-parts of the filled-in DRET template), specified requirements (SPLACID specifications), V&V requirements (results obtained by using the Alloy Analyzer tool).

FUTURE RESEARCH DIRECTIONS

The PRISMA process, which has been presented in this chapter, offers a methodological support to engineer adequate transaction models for cloud computing by customizing their ACIDity qualitatively.

Thanks to the identification of the building properties (sub-properties) which, in turn, characterize the ACID properties, it is possible to evaluate qualitatively the ACIDity of one model with respect to another one, simply by considering which sub-properties it is composed of. This evaluation capability suggests a systematic organization of transaction models into a lattice structure, which holds together many individual elements, otherwise unstructured, into one coherent and understandable order. This evaluation capability also pioneers the path to the provision of a quantitative evaluation of ACIDity.

As discussed throughout the chapter, since cloud computing is subjected to the CAP theorem, the customization of ACIDity is necessary and has to be planned carefully. To enforce the PRISMA process effectiveness in offering a valid means, it would be useful to have at disposal metrics to evaluate quantitatively the ACIDity of a transaction model. These metrics could be used to measure the ACIDity of the transaction models currently in use in the context of cloud computing so that to achieve a range of reliability.

To provide a quantitative metric for the ACIDity, the number of histories generated by the Alloy Analyzer tool, within a certain scope, could be counted. The ACIDity of two transaction models might, therefore, be compared on the basis of the number of histories allowed. For instance, with the predicate "moreACIDthan (transaction model A, transaction model B, application X)" a truth value might be assigned by verifying that the number of histories allowed by the transaction model A structuring the application X is less than the number of histories allowed by the transaction model B structuring the same application. More granularly, it would be useful to be able to establish the role of each single feature that composes a transaction model in decreasing or increasing the ACIDity of the model itself (that is, the contribution of each sub-property in increasing/decreasing the number of allowed histories).

The work presented in (Prömel, Steger, & Taraz, 2001) could represent another possibility for counting the histories.

The percentage of ACIDity characterizing a computation structured according to a transaction model could be calculated as follows:

ACIDity% = (#forbiddenHistories)/(#allpossibleHistories),

where the number of forbidden histories is obtained by subtracting to the number of all possible histories the number of allowed histories.

By decreasing the ACIDity, that is by choosing base like features, at some point a transaction model degenerates completely since its ACIDity is neutralized. A threshold that separates feasible and reasonable transaction models from non-reasonable, neutralized transaction models must exist. A metric would allow this threshold to be identified. An ACIDity metric would allow an engineer to quantify and plan the ACIDity and as a logical consequence this would allow the loss in terms of global consistency to be planned as well.

CONCLUSION

The technological change represented by cloud computing forces cascading changes and, as a consequence, several issues need to be solved to identify and adequately engineer the cascading changes. Since the quality and, in particular, reliability remains a fundamental requirement and since transaction models represent effective means to increase the quality of a computation, this chapter has been devoted to the identification of the cascading changes that involve transaction models and, more specifically, their requirements in terms of Atomicity, Consistency, Isolation and Durability. To engineer the changes, then this chapter has introduced a requirements engineering process, called PRISMA. This process integrates a product line perspective and supports engineers to reuse systematically properties of pre-existing transaction models to achieve adequate transaction models, which meet the application domain's requirements as well as the technological domain's requirements. Finally, on the basis of the current understanding, the PRISMA process has been applied to engineer Atomicity-related reusable assets. This chapter has provided a first step to achieve adequate transaction models for cloud computing.

REFERENCES

Adya, A., Liskov, B., & ÓNeil, P. (2000). Generalized isolation level definitions. *16th International Conference on Data Engineering(ICDE)* (ss. 67–80). San Diego, California, USA: IEEE Computer Society.

Alloy Analyzer 4. (u.d.). Retrieved from: http://alloy.mit.edu/alloy4/ den 25 November 2010

Avizienis, A., Laprie, J. C., Randell, B., & Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing, 1* (1), 11–33.

Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., & O'Neil, P. (1995). A critique of ANSI SQL isolation levels. *ACM SIGMOD international conference (ACM Special Interest Group on Management of Data). 24.* San Jose, California, USA: ACM Press.

Bernstein, P. A., & Newcomer, E. (2009). *Principles of Transaction Processing, Second Edition.* 30 Corporate Drive, Suite 400, Burlington, MA 01803 USA: Morgan Kaufmann Publishers.

Besancenot, J., Cart, M., Ferrié, J., Guerraoui, R., Pucheral, P., & Traverson, B. (1997). *Les systèmes transactionnels : concepts, normes et produits.* Paris, France: Editions Hermès.

Brewer, E. A. (2000). Towards robust distributed systems (abstract). *the 19th Annual ACM Symposium on Principles of Distributed Computing* (s. 7). July 16-19, Portland, Oregon: ACM.

Buyya, R., Yeo, C. S., Venugopal, S., Broberg, J., & Brandic, I. (2009). Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility. *Future Generation Computer Systems , 25* (6), 599-616.

Chrysanthis, P. K. (1991). *A Framework for Modeling and Reasoning about Extended Transactions.* Doctoral Dissertation, University of Massachusetts, Department of Computer and Information Science, Amherst, Massachusetts.

Clements, P., & Northrop, L. (2001). *Software Product Lines: Practices and Patterns.* Reading, MA, USA: Addison Wesley.

Derks, W., Dehnert, J., Grefen, P. W., & Jonker, W. (2001). Customized atomicity specification for transactional workflows. *Third International Symposium on Cooperative Database Systems for Advanced Applications (CODAS)* (ss. 140-147). Beijing, China: IEEE Computer Society.

Drew, P., & Pu, C. (1995). Asynchronous consistency restoration under epsilon serializability. *28th Hawaii International Conference on System Sciences (HICSS)* (ss. 717–726). Kihei, Maui, Hawaii, USA: IEEE Computer Society.

Elmagarmid, A. K. (1992). *Database Transaction Models for Advanced Applications.* The Morgan Kaufmann Series in Data Management Systems.

Gallina, B. (2010). *PRISMA: a Software Product Line-oriented Process for the Requirements Engineering of Flexible Transaction Models.* Laboratory for Advanced Software Systems, University of Luxembourg, Luxembourg: Unpublished doctoral dissertation. Retrieved November 16, 2010.

Gallina, B., & Guelfi, N. (2008). A Product Line Perspective for Quality Reuse of Development Framework for Distributed Transactional Applications. *2nd IEEE International Workshop on*

Quality Oriented Oriented Reuse of Software (QUORS), co-located with COMPSAC (ss. 739-744). Turku, Finland: IEEE Computer Society, Los Alamitos, CA, USA.

Gallina, B., & Guelfi, N. (2007). A Template for Requirement Elicitation of Dependable Product Lines. *13th International Working Conference on Requirements Engineering: Foundation for Software Quality* (ss. 63-77). Trondheim, Norway: Lecture Notes in Computer Science.

Gallina, B., & Guelfi, N. (2008). SPLACID: An SPL-oriented, ACTA-based, Language for Reusing (Varying) ACID Properties. *In 32nd Annual IEEE/NASA Goddard Software Engineering Workshop (SEW-32)* (ss. 115–124). Porto Sani Resort, Kassandra, Greece: IEEE Computer Society.

Gallina, B., Guelfi, N., & Kelsen, P. (2009). Towards an Alloy Formal Model for Flexible Advanced Transactional Model Development. *In 33rd International IEEE Software Engineering Workshop (SEW-33).* Skövde, Sweden: IEEE Computer Society.

Garbus, J. R. (den 2 February 2010). In-Memory Database option for Sybase Adaptive Server Enterprise. *Database Journal* .

Gilbert, S., & Lynch, N. (June 2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News , 33* (2), ss. 51-59.

Gray, J. (1980). A Transaction Model. i J. W. de Bakker, & J. van Leeuwen (Red.), *In Proceedings of the 7th Colloquium on Automata, Languages and Programming. 85*, ss. 282-298. Noordweijkerhout, The Netherland: Lecture Notes in Computer Science. Springer Verlag.

Gray, J., & Reuter, A. (1993). *Transactions Processing: Concepts and Techniques.* Morgan Kaufmann Publishers.

Hohpe, G. (2009). *Into the Clouds on New Acid.* Retrieved from: Enterprise Integration Patterns: http://www.eaipatterns.com/ramblings/68_acid.html den 2 November 2010

Härder, T., & Reuter, A. (1983). Principles of Transaction-Oriented Database Recovery. *ACM Computing Survey , 15* (4), 287-315.

Jackson, D. (2006). *Software Abstractions: Logic, Language, and Analysis.* MIT Press.

Jackson, M. (1998). *Software Requirements & Specifications.* Addison-Wesley.

Karlapalem, K., Vidyasankar, K., & Krishna, P. R. (2010). Advanced Transaction Models for e-Services. *In 6th World Congress on Services, Tutorials, services* (ss. xxxii-xxxiii). IEEE Computer Society.

Kiringa, I. (2001). Simulation of advanced transaction models using GOLOG. *In Revised Papers from the 8th International Workshop on Database Programming Languages. 2397*, ss. 318-341. Frascati, Italy: Lecture Notes in Computer Science, Springer-Verlag.

Klaus, P., Böckle, G., & van der Linden, F. J. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques (1 edition).* Springer-Verlag.

Kuloor, C., & Eberlein, A. (2002). Requirements engineering for software product lines. *15th International Conference on Software and Systems Engineering and their Applications (ICSSEA).* Paris, France.

Levy, E., Korth, H. F., & Silberschatz, A. (1991). A theory of relaxed atomicity (extended abstract). *the Tenth Annual ACM Symposium on Principles of Distributed Computing (PODC)* (ss. 95-110). Montreal, Quebec, Canada: ACM, New York, NY.

Moss, J. E. (1981). *Nested Transactions: An Approach to Reliable Distributed Computing.* PhD dissertation, Massachusetts Institute of Technology, USA.

OMG. (2008). *Software & systems Process Engineering Meta-model (SPEM), v 2.0. Full Specification formal/08-04-01.* Object Management Group.

Pritchett, D. (2008). BASE: An Acid Alternative. *ACM Queue , 6* (3), 48-55.

Prömel, H. J., Steger, A., & Taraz, A. (2001). Counting partial orders with a fixed number of comparable pairs. *Combinatorics, Probability and Computing. 10.* Cambridge University Press.

Puimedon, A. S. (2009). *Transactions for grid computing, Advanced e-Business Transactions for B2B-Collaboration seminar, University of Helsinki.* Retrieved from: www.cs.helsinki.fi/group/.../Transactions_grid_computing.pdf den 02 November 2010

Ramamritham, K., & Chrysanthis, P. K. (1996). A taxonomy of correctness criteria in database applications. *The International Journal on Very Large Data Bases (The VLDB Journal) , 5* (1), 85-97.

Sadeg, B., & Saad-Bouzefrane, S. (2000). Relaxing correctness criteria in real-time DBMSs. i S. Y. Shin (Red.), *ISCA 15th International Conference Computers and Their Applications* (ss. 64–67). New Orleans, Louisiana, USA: ISCA.

Walborn, G. D., & Chrysanthis, P. K. (1995). Supporting semantics-based transaction processing in mobile database applications. *In the 14 th IEEE Symposium on Reliable Distributed Systems,, 13-15 September* (s. 31). Bad Neuenahr, Germany: IEEE Computer Society, Los Alamitos, CA, USA.

Webber, J., & Little, M. (u.d.). *Introducing ws-coordination*. Retrieved from: http://www2.syscon. com/itsg/virtualcd/webservices/archives/0305/little/index.html den 02 November 2010

Withey, J. (1996). *Investment Analysis of Software Assets for Product Lines.* Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

Vogels, W. (2008). Eventually Consistent. *Queue , 6* (6), 14-19.

ADDITIONAL READING SECTION

Abadi, D. J. (2009). Data Management in the Cloud: Limitations and Opportunities. *IEEE Data Engineering Bulletin , 32* (1), 3-12.

Bernstein, P. A., Hadzilacos, V., & Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems.* Addison Wesley.

Burrows, M. (2006). The Chubby lock service for loosely-coupled distributed systems. *7th symposium on Operating Systems Design and Implementation (OSDI)* (ss. 335-350). Berkeley, CA, USA: USENIX Association.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., o.a. (2006). Bigtable: a distributed storage system for structured data. *the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI). 7*, ss. 205-218. Berkley, CA, USA: USENIX Association.

Cooper, B. F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H., o.a. (2008). PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment , 1* (2), 1277-1288.

Ericson, K., & Pallickara, S. (2010). Survey of Storage and Fault Tolerance Strategies Used in Cloud Computing. i B. Furht, & A. Escalante (Red.), *Handbook of Cloud Computing* (ss. 137-158). Springer Verlag.

Fox, A., Gribble, S. D., Chawathe, Y., Brewer, E. A., & Gauthier, P. (1997). Cluster-based scalable network services. i W. M. Waite (Red.), *Sixteenth ACM symposium on Operating systems principles (SOSP)* (ss. 78-91). New York, NY, USA: ACM.

Guo, H., Larson, P., & Ramakrishnan, R. (2005). Caching with "good enough" currency, consistency, and completeness. *31st international conference on Very large data bases (VLDB)* (ss. 457-468). VLDB Endowment.

Helland, P. (2007). Life beyond Distributed Transactions: an Apostate's Opinion. *Third Biennial Conference on Innovative Data Systems Research*, (ss. 132-141). Asilomar, CA, USA.

Kraska, T., Hentschel, M., Alonso, G., & Kossmann, D. (2009). Consistency Rationing in the Cloud: Pay only when it matters. *Proceedings of the VLDB Endowment (PVLDB) , 2*, 253-264.

Stonebraker, M., Madden, S., Abadi, D. J., Harizopoulos, S., Hachem, N., & Helland, P. (2007). The End of an Architectural Era (It's Time for a Complete Rewrite). *33rd International Conference on Very Large Data Bases (VLDB)* (ss. 1150–1160). University of Vienna, Austria: ACM.

Tanenbaum, A. S., & Van Steen, M. (2006). *Distributed Systems: Principles and Paradigms.* Prentice Hall, 2 edition.

Weikum, G., & Vossen, G. (2002). *Transactional Information Systems.* Morgan Kaufmann.

KEY TERMS & DEFINITIONS

State - This term identifies a mapping from storage unit names to values storable in those units.

Object (or or data-item) - This term identifies a single pair <name, value>.

Consistency constraint - This term identifies a predicate on objects. A state satisfying all the consistency constraints defined on objects is said to be consistent.

Operation - This term identifies the access of a single object. Two types of access are allowed. One type identifies operations that read (get the value). The other type identifies operations that write (set the value).

Event - This term identifies the execution of a single operation on the state.

Work-unit - This term identifies the set of possible executions of a partially ordered set of logically related events.

History - This term identifies the set of possible complete executions of a partially ordered set of events belonging to a set of work-units.

Transaction type - This term identifies a specific set of properties that have to be satisfied by work-units. The set of properties constrains the events belonging to a work-unit.

Transaction model - This term identifies the type of structure/ordering used to organize the transaction types that are used to decompose a computation. This structure identifies a multigraph. Nodes are identified by transaction types and edges by structural dependencies existing among the transaction types. The structure/ordering that organizes the transaction types has an impact on the ordering of events belonging to a history. A containment dependency between two transaction types, for instance, has the following impact on the history: all the events belonging to a work-unit of type "content" are enclosed within those events that represent the boundary and that belong to a work-unit of type "container".