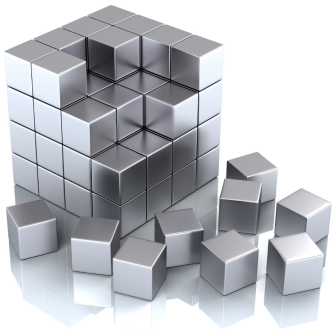
TÉCNICO LISBOA



A Development Framework for Normalized Systems

Pedro Filipe Pires Simões

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Diogo Manuel Ribeiro Ferreira

Examination Committee

Chairperson: Prof. Mário Rui Fonseca dos Santos Gomes

Supervisor: Prof. Diogo Manuel Ribeiro Ferreira

Members of the Committee: Prof. Paulo Jorge Fernandes Carreira

June 2016

Resumo

Os sistemas de informação têm de evoluir gradualmente ao longo do tempo de forma a responder à constante mudança de requisitos e circunstâncias em que os mesmos são inseridos. Esta volatilidade provoca não só um aumento contínuo da complexidade arquitetural como também uma degradação da qualidade de software. A teoria dos Sistemas Normalizados aborda este problema através da sistematização do desenvolvimento de sistemas de informação com o objetivo de capacitar os mesmos com a referida capacidade de evolução, garantindo que cada nova funcionalidade introduzida possa ser implementada através de um número limitado de alterações que não dependem do tamanho do sistema. A teoria é suportada por uma arquitetura de software que inclui um conjunto de cinco elementos distintos. Um desses elementos é o elemento Workflow, que representa o comportamento de um processo de negócio com base em máquinas de estado. Esta dissertação propõe uma nova arquitetura de software para Sistemas Normalizados que se baseia na separação entre uma perspetiva estrutural e uma perspetiva comportamental. Na perspetiva comportamental, são usadas regras e eventos como elementos granulares que podem ser combinado para formar padrões de comportamento complexo, garantindo ao mesmo tempo uma baixa interdependência entre esses elementos. Embora a arquitetura proposta seja agnóstica em relação à tecnologia subjacente, nesta dissertação foi desenvolvido um protótipo baseado na plataforma .NET e linguagem C#. Esta implementação inclui também uma extensão ao ambiente de desenvolvimento Visual Studio, a fim de fornecer um conjunto de ferramentas de auxílio ao desenvolvimento, usando geração de código.

Palavras-chave: Arquiteturas de Software, Sistemas Normalizados, Workflow, Processos de Negócio, Regras ECA

Abstract

Information systems need to evolve continuously to adapt to the varying requirements and circumstances of the dynamic environment in which they run. This volatility gradually leads to an increase in architectural complexity and a decrease in software quality. The theory of Normalized Systems addresses this problem by developing information systems that can evolve over time, guaranteeing that each new feature can be implemented with a limited number of changes that does not depend on the actual size of the system. The theory is supported by a software architecture that includes a set of five distinct elements to implement the application logic. One of those elements is the workflow element, which represents a sequence of actions implementing process behavior based on state machines. This dissertation proposes a new architecture for Normalized Systems that is based on a structural perspective and a behavioral perspective. In the behavioral perspective, rules and events are used as fine-grained elements to build complex behavior, while guaranteeing low coupling between all elements. Although the proposed architecture is agnostic with respect to the underlying technology, a prototype implementation based on the .NET Framework and the C# language was developed. This implementation also extends Visual Studio in order to provide a set of development plugins and advanced code-generation capabilities.

Keywords: Software Architectures, Normalized Systems, Workflow, Business Processes, ECA Rules

Contents

List of Figures

List of Tables

Listings

Acronyms

AVT Action Version Transparency

BPML Business Process Modeling Language

BPMN Business Process Model and Notation

BPMS Business Process Management System

CLI Command Line Interface

COM Component Object Model

DFD Data Flow Diagram

DLL Dynamic-Link Library

DVT Data Version Transparency

ECA Event-Condition-Action

IDE Integrated Development Environment

IS Information System

J2EE Java Platform, Enterprise Edition

LINQ Language-Integrated Query

LSP Liskov Substitution Principle

NS Normalized Systems

POS Point-Of-Sale

RAD Rapid Application Development

SDK Software Development Kit

SoC Separation of Concerns

SoS Separation of States

T4 Text Template Transformation Toolkit

UML Unified Modeling Language

VS Visual Studio

WPF Windows Presentation Foundation

XML eXtensible Markup Language

XSD XML Schema Definition

XSLT eXtensible Stylesheet Language for Transformation

Chapter 1

Introduction

The problem of software degradation is well known in software maintenance and development, and it was stated as early as 1968 [2]. It almost seems that software can actually age like a living being. As Parnas [3] has stated, one sign that Software Engineering has matured will be when software engineers shift their primary concern to focus on the long-term health of the Information Systems instead on only its first release.

While software does not age in the same literal way, it is clear that it degrades over time. This degradation will reflect mainly on the software structure leading it into to a code base that is hard to maintain and evolve. A significant contribution to this software ”aging” is the fact that organizations are increasingly dependent on Information Systems to perform their day-to-day operations, and these operations must frequently adapt their requirements considering that organizations must compete with each other and evolve according to the needs of the market. These frequent adjustments on a Information System (IS) requirements are related to the first law of evolvability [4], where an IS built with a specified set of requirements at a certain point in time and according to a particular reality, over time it will diverge continuously from the contemporary reality. Therefore, an IS will gradually become less useful.

The attempt to cope with the first law of evolvability will naturally lead to the second law of evolvability. This law states that frequent changes to software will cause an increase in its complexity up to an unmanageable state unless some effort is made to counteract this tendency. This trend will bring maintainability and evolvability into a continuous reduction, and also it will be experienced a functionality decrease. Additionally, the resistance to change will increase in high numbers, and there is also a growth in the cost (time and money) of maintaining an IS over its lifetime.

These tendencies or decay processes will endure to a point where it becomes more efficient to replace the IS with a new, recreated version of the original IS. This approach brings another set of problems. The most challenging one being the compliance that the new IS will have to ensure regarding with the requirements of the original IS since its inception. Additionally, the original IS will evolve independently during the time that a new project starts until its release. Therefore, this new IS project will suffer since its start from continuous requirements changes in addition to the full set of legacy requirements. This approach should be rethought, considering that it does not classify as the best solution for this

problem by the previously presented reasons.

The ideal approach to this problem is the development of an IS which is capable of evolving since its inception, and whose maintenance and evolution cost does not increase uncontrollably over time. That is precisely the goal of Normalized Systems (NS) [5].

Changes which generate an impact that depends on the size of the system are regarded as unstable and are referred to, in NS theory, as *combinatorial effects*. The theory of NS addresses the problem of developing information systems that can evolve over time, by guaranteeing that each new feature can be implemented with a limited number of changes that does not depend on the actual size of the system. This stability ensures that the cost of an IS application project will not grow uncontrollably.

1.1 Problem

NS theory is well documented in the book [5] by its authors. Unfortunately, only some technical descriptions are present and are scarce in their explanation on how to transform the theory into an IS application project. Also, there is not any software development framework publicly available, capable of introducing NS theory concepts into an IS application project. For these reasons the first challenge of this work is presented below:

Question 1. *Can a software development framework be implemented using NS theory as its foundations, exhibiting the same system theoretic stability as intended by the NS original architecture?*

Considering this first challenge, the principal goal of this work is not the creation of a competitive and alternative framework to the original one, neither a one-to-one recreation of it. Instead, it is the design of a new independent architecture based on the conceptual principles of the NS theory and a software framework developed from scratch using only the concepts as its architectural foundations.

The NS theory is supported by a software architecture that includes a set of five distinct elements to implement the application logic. NS theory ensures stability and evolvability with respect to a set of anticipated changes to these fine-grained elements.

One of those elements in the NS software architecture is the Workflow element, which allows the embedding of process logic in a normalized system. Traditionally, this process logic is specified as a state machine that defines a sequencing of Actions, and where the execution of each Action brings the process from a previous state into a new state.

In this work, some of the fundamental principles on which NS theory is built will be used as an inspiration for the design of the proposed framework, namely:

- The principle of having a *fine-grained modular structure* which can accommodate changes without requiring an extensive system overhaul.

- The principle of having *low coupling* and *high cohesion* between elements so that a change in one element has the least potential to generate impact on other elements.

These principles influenced the design of the proposed architecture process logic into a rule-based workflow approach using ECA rules. This approach separates process logic from application logic into a fine-grained software element structure. Additionally, it also introduces an increase in the process logic flexibility, where more workflow patterns can be represented by using this rule-based approach.

The idea of using Event-Condition-Action (ECA) rules to define and implement process behavior goes back to the 1990s, with pioneering works such as TriGSflow [6], WIDE [7], SWORDIES [1], and EVE [8]. Since then, several authors have proposed and discussed the benefits of using ECA rules for describing and implementing business processes [9, 10, 11]. Among these benefits, there are some characteristics of ECA rules that are widely recognized and that are of special interest in the context of normalized systems:

- ECA rules are inherently flexible in the sense that they can be changed or adapted to new requirements, and it might even be possible to do this without interrupting running processes.
- ECA rules deal with error handling and exceptions in a natural way. Since errors and exceptions are also events, the rules to handle them can be defined in a similar way to any other rule.
- In traditional graph-based approaches, the transition to a new state usually occurs upon completion of an activity. However, with a rule-based approach it is possible for a single activity to be the source of multiple events, with new rules being fired at any point during execution.

Basically, an ECA rule has the following form:

ON *event* **IF** *condition* **THEN** *action*

This means that when a certain event occurs, a condition is evaluated and, if the condition is true, some action is invoked. When the action raises an event, another rule will be fired, creating a chain of actions and events that is the basis for implementing sequential behavior.

For these reasons, the following challenge is presented below:

Question 2. *Can a software development framework based on NS theory continue to exhibit evolvability and stability when using a rule-based approach and comply with all the original counterpart theoretical basis?*

1.2 Objectives

The objectives of this work start with an analysis of the NS original theory using the book of this topic [5], complemented by other related articles. Other theories, which have a complementary nature to this subject, will also be studied with the objective to analyze the original theory from different perspectives.

After this introspection, this work has the pretends to extend the original NS theory architecture with the use of “rule-event” based elements resulting in an independent architecture based on the NS theory concepts and rules.

With this approach is expected to achieve an improved fine-grained design, leveraged by the introduction of the behavioral perspective into the architecture will increase the refinement of the software elements.

It is pretended to implement a base framework in the C# programming language, with the objective of extending and generating C# code. The Visual Studio (VS) will be extended with integrated tools to ease the design and implementation of IS applications. This extension is supposed to expand the framework descriptors into ready to build C# code and eliminate the task of writing manually the error-prone, non-functional and systematic source-code needed to assemble the complete application architecture.

With the proposed framework it should be possible to build a variety of IS types in an organization, exhibiting high evolvability and low maintenance costs. Organizations and their software developers should be able to focus only on the problem to solve instead on all other structural and cross-cutting concerns, which are taken care of by the framework. The built IS will inherit a normalized systems software architecture "out of the box".

Finally, this framework and tools should be capable of designing an application containing multiple element versions and maintaining stability and evolvability at the same time. For this conceptual verification, it will be used a Retail industry case study focused on promotions and customer loyalty, considered a theme that exhibits a highly dynamic environment.

1.3 Outline

This work comprises the following chapters:

- Chapter 1 - *Introduction*: In this chapter, it is first introduced the background context of this dissertation, followed by the problem statement that this work proposes to solve. Finally, it is presented the objectives section where it is explained what is the result of the dissertation work and how this result answers the questions that were stated in the previous section.

- Chapter 2 - *Normalized Systems*: Presents an overview of this work related literature and background theory targeted on the study and analysis of the original theory.

- Chapter 3 - *Proposed architecture*: Presents the proposed principles based on the analysis of the original theory augmented by complementary insights, followed and concluded by the definition of this dissertation proposed framework architecture.

- Chapter 4 - *Implementation*: Explains the distinct layers that a NS project has, from the base library, expanded code and finally through the custom code. The base framework is described in detail using a very simple "hello world" example. Regarding to the design-time features, starts by explaining how the software elements are defined in a file descriptor, designed in VS, expanded, checked and finally compiled. It is also described how the integration with VS is made, and how this tool can assist the developers in their project development.

- Chapter 5 - *Case study*: The product of this dissertation will be validated in its correctness and stability against a case study inspired and related to a retail industry real world scenario. This case study is divided step-by-step, simulating the continuous change of projects requirements that a project like this can suffer, and at the same time analyzing how the framework will tackle the encountered problems.
- Chapter 6 - *Conclusion*: In this chapter, it is summarized this dissertation work evaluation, achievements, difficulties and possible future evolution.

Chapter 2

Normalized Systems

Normalized Systems (NS) is a software engineering approach that is being studied and developed at the University of Antwerp, Belgium, since 2009. The authors of the theory wrote a book [5] focused to the theory presentation and explanation, but not before they made a complete and detailed overview of the major Information Systems (IS's) constructs and methodologies that have been proposed in theory and practice.

In this overview, the authors identified important issues in three distinct domains:

- *Vagueness*: In software development, there are multiple ways to accomplish the same goal, all of them considered to be correct taking into account their architecture design. For instance, the concept of "low coupling" can be approached in slightly different ways. This diversity of opinions and solutions in how to tackle a particular problem indicates an intrinsic vagueness in software development.

- *Lack of systematic application*: Some project management constraints, such as time and budget, can influence the consideration of reuse and evolvability as a primal concern in the IS project design. Additionally, the uncertainty that an anticipated change will ever occur at all, can also shift the IS project focus into the first release instead of the evolvability and maintainability of the project. These reasons can explain the lack of use of a proper, correct and stable architecture design in IS projects. It is tough, challenging and costly to build an IS architecture that exhibits evolvability since its inception towards many anticipated changes. It can be achieved by implementing an IS project based on a fine-grained modular structure that limits the impact of every anticipated change.

- *Limited traceability*: It is desirable that the mapping between the real world and the modular structure of a IS architecture should be closest to a one-to-one correspondence. This mapping would provide traceability between this two layers and by consequence assist in the evolvability of a IS taking into account that a change in the real world can be mapped directly to a change in the IS architecture. This type of one-to-one mapping cannot be realized by current methodologies. Nowadays, the mapping between the real world and IS architecture is complicated and cumbersome, having mappings at different levels and nature. For these reasons, it will be extremely hard

for an organization to align its IS with its volatile business context.

NS addresses these issues and the required ability of contemporary organizations to react quickly to an ever-changing and volatile environment to gain or maintain competitivity [12]. Due to these continuously changing business requirements, NS identifies *evolvability* [13] as one of the main concerns in the design of IS'ss.

Lehman's laws of software evolution [4] are often invoked in connection with NS theory. For example, the Lehman's second law says that an increase in architectural complexity (and a subsequent decrease in software quality) will arise if no effort is made to reduce or maintain that complexity. As a result, the continuous changes made to an IS deteriorates and erodes its structure over time.

NS theory has two fundamental concepts or starting points:

- *Systems theoretic stability*: where a bounded input function results in limited output values for an infinite time;
- *Assumption of unlimited system evolution*: considering an unlimited period and an unlimited evolution of the system, the primitives and the number of dependencies between them become infinite or unbounded for an infinite time;

NS theory relies on the concept of *stability* to support information systems evolvability and to counter the effects of Lehman's law of increasing complexity [14]. In NS systems theoretic stability means that a bounded set of changes must cause a bounded set of impacts, even as the size of the system grows up to infinity [15, 16].

In this context, Lehman's first law says that a system must be continually adapted or it becomes progressively obsolete. NS theory sees this continuous adaptation in the light of unlimited systems evolution [17], where the number of software primitives and the corresponding number of dependencies between them must be kept under control as the system grows. Otherwise, as the size of the system tends to infinity, they will become *unbounded*, leading the IS project into a unstable state.

Therefore, the impact i of a system change c must not depend on the size of the system n, otherwise $i \to \infty$, i.e. i becomes unbounded. Changes which generate an impact that depends on the size of the system are regarded as unstable and are referred to, in NS theory, as *combinatorial effects*. According to NS theory, the occurrence of combinatorial effects will adversely affect evolvability. NS is essentially an attempt to systematically eliminate combinatorial effects.

Specifically, NS theory proposes a set of design principles, intended to act as design rules to identify and circumvent most combinatorial effects, whereby an information system becomes harder and costlier to change as it grows over time [18]. In order to prevent combinatorial effects, NS theory advocates the use of a fine-grained and modular structure, by considering that the core issues regarding evolvability relate to a correct design of modular structures and substructures. Also, these principles come together with the proposal of a particular platform-independent application architecture, which facilities the implementation and enforcing of such principles in practice.

The NS theory presents gradually its architectural proposal with two design models: a higher abstraction level and simpler model describing the software *entities* present in the NS theory; and a more

complex and detailed model describing the software *elements* of the NS theory. In the next section describes the basic elements of such architecture.

2.1 Software entities

The architectural model proposed by NS theory involves software elements which are instances of a typified set of *software entities*. NS theory regards the most generic and high-level view of the modular structure of an information system as actions processing data. Therefore, the main software entities are *data entities* and *action entities*:

- *Data entities*: are software entities that contain primitive fields and other data entities reference fields. Holding business data is their main concern.
- *Action entities*: are software entities that represent a task at a given modular level in an hierarchy. An action entity consumes and produces data entities as input and output.

In this context, a task is assumed to correspond to a single change driver, and tasks can be divided into functional and supporting tasks [19]:

- A *functional task* is a task that performs a specific functional operation in an IS.
- A *supporting task* is a generic task that performs a cross-cutting concern in an IS.

Cross-cutting concerns traverse tasks in a way that introduces change drivers inside them. In a certain way, cross-cutting concerns will pollute the core concern of these tasks. Therefore, if each task is to be mapped to a single change driver, these cross-cutting concerns must be separated from other tasks, and that is the reason why they are defined as supporting tasks.

To perform a task it may be necessary to import software entities from other systems, introducing dependencies to and from these systems in an implicit way. The presence of one or more software entities that belong to another system inside a task is defined as an *external technology*. An external technology inside an action entity will always be mapped to a different task, in order to achieve some degree of isolation from the evolution of such external technology. An external technology is identified as an independent change driver within the action entity. Therefore, it must be a separate task within the action entity.

The concept of a change driver, that can evolve independently from other change drivers, identifies separates tasks and implies that a change driver cannot contain other change drivers. Therefore, NS theory does not allow tasks within tasks.

2.2 Stability of software entities

The main concept in NS theory is system stability, where a bounded set of changes results in a limited set of impacts to the system, even as the size of the system tends to infinity [20]. However, NS theory is

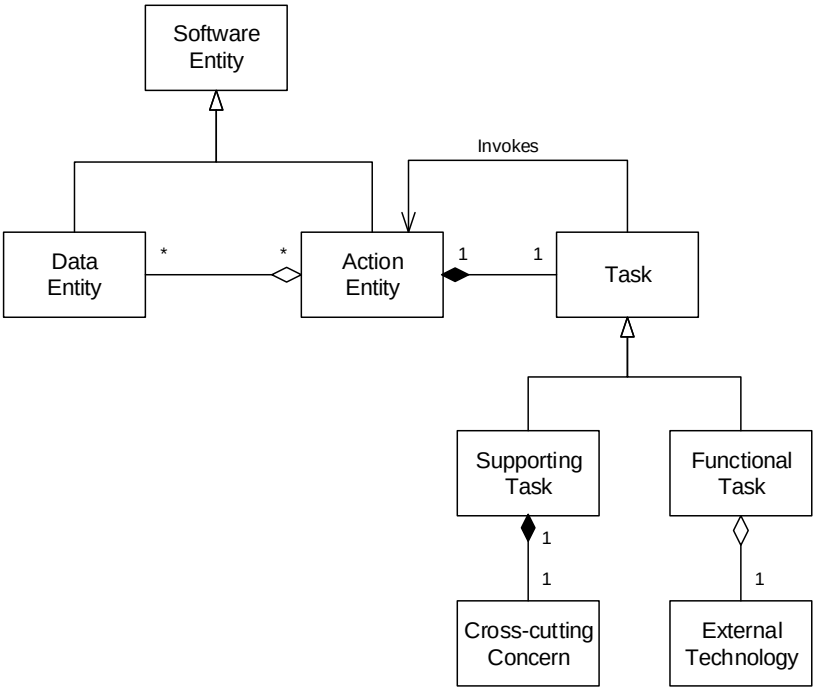


Figure 2.1: Conceptual model of software entities in NS theory

not so ambitious as to try to guarantee system stability under any sort of possible change to the system; that is probably impossible to achieve. Instead, NS theory aims to ensure system stability with respect to a set of anticipated changes, namely:

- The introduction of an additional data field in a data entity: does not introduce impact on any other software entity because this new field will not alter the calling interface or reference of any software entity.
- The introduction of an additional data entity: does not introduce impact on any other software entity, considering that prior of this addition no other software entity had a reference to it.
- The introduction of an additional action entity: does not introduce impact on any other software entity, considering that prior of this addition no other software entity had a reference to it.
- The introduction of an additional version of a task: does not introduce impact on any other software entity, considering that existing software entities that have a reference to the task will continue to reference the old version instead of the new version.

These anticipated changes ensure that new versions can be added for all action entities, data entities and tasks. Combinatorial effects are contained and avoided on these anticipated changes, resulting on the maintenance of the IS stability, even as the size of the system grows.

It should be noted that only additions are defined on this list of anticipated changes. Modifications are regarded as refactoring and can be seen as the combination of an addition followed by a deletion. Deletions are considered a matter of *garbage collection*: deletion is defined as an operation that seeks

for all existing references to software entities followed by the exclusion of those that are not referenced by any other software entity. There is no explicit need to delete these parts since without any reference to other software entity, nothing will use them. If the system is stable, unused parts of the system have no impact on the rest of the system.

2.3 NS design theorems

NS theory defines four theorems regarding the development of information systems that are stable with respect to the set of anticipated changes above. These four theorems are the cornerstones of NS theory.

Separation of Concerns

Theorem 1. *Separation of Concerns (SoC) : An action entity can only contain a single task (single change driver).*

In NS theory, separation of concerns implies that every change driver should be isolated into its own modular construct [21]. Since a change driver cannot contain another change driver, and a task must have a single change driver, an action entity can only contain a single task.

Concerning external technologies, since an external technology is an independent change driver and considering that a change driver will indicate having a separate task, an action entity can only contain a single external technology.

Cross-cutting concerns introduce change drivers inside tasks and are considered to be different change drivers. Therefore, they must constitute separate tasks by themselves, in this case supporting tasks. These supporting tasks require separate action entities for them as well.

Data Version Transparency

Theorem 2. *Data Version Transparency (DVT) : Data entities that are received as input or produced as output by action entities need to exhibit version transparency.*

A fundamental concept in NS theory is *version transparency*, which states that a software entity can have multiples versions of itself without affecting other software entities. Version transparency applies both to data entities and to action entities.

The theorem of data version transparency states that data entities can have multiple versions without affecting action entities that consume or produce them as input/output and also other data entities that have a reference to them. One consequence from this theorem is that a separate data entity is needed to wrap and encapsulate data entities representing different versions of the data entity.

Another consequence from this theorem is that when an action entity interacts with a data entity, it must interact through some kind of mechanism that allows the data entity to evolve without having an impact on the action entity. For example, this can be achieved by using property get and set methods. This principle is closely related to information hiding in object-oriented programming [22]. This is a

common practice to implement a generic stable object interface by protecting the inner working parts of a class.

Cross-cutting concerns for data entities must also be implemented in separate action entities, as implied by the first theorem, but must be implemented at the level of the data entity that wraps all versions and not at the level of their individual versions.

Action Version Transparency

Theorem 3. *Action Version Transparency (AVT) : Action entities that are called by other action entities need to exhibit version transparency.*

The theorem of action version transparency focuses on the interaction between action entities with other action entities. Since a task can have multiple versions and an action entity maps directly to a single task, this implies that action entities can have multiple versions. They must implement version transparency.

Version transparency on actions entities means that an action entity can have multiple versions without affecting other action entities that invoke them. One consequence from this theorem is that a separate action entity is needed to represent each different version.

Cross-cutting concerns for action entities must also be implemented in separate action entities, as implied by the first theorem, but must be implemented at the level of the action entity that wraps all versions and not at the level of their individual versions.

Separation of States

Theorem 4. *Separation of States (SoS) : The calling of an action entity by another action entity needs to exhibit state keeping.*

Synchronous calls constituted by stacks of objects invoking other objects is a typical pattern used in software development. This relationship results in a hierarchy of objects with a considerable dependency between them. The existence of this hierarchy increases the chances of combinatorial effects.

The theorem of separation of states, which focuses on the invocation of action entities, says that when an action entity calls another action entity, this calling must be stateful. In other words, every time an action entity is called, its state must be kept. This theorem introduces the need to define *action states* in order to keep track of each action entity call.

A manifestation of this theorem is that separation of states theorem closely relates to stateful workflows where intermediate states are kept, and action entities within the workflow react to consecutive states, chaining their execution into a processing similar to a workflow.

One consequence from this theorem is that the state of an action entity has to be kept for every call of that action entity. This state can be part of the data entity that is used as target of this action entity, meaning that the execution of action entities can be stored in the target data entity of a workflow. This can be used in such a way that a particular state will trigger a specific action entity of the workflow.

Action entities themselves must be unaware of this state keeping, since this it is not their main concern. In other words, the control of an action invocation must be external to them.

With action states, exception handling does not work by throwing the error up the call stack but by registering it in the action state, therefore allowing another action entity to react to it. One advantage of this principle is the ability to process action entities in an asynchronous way. On current multi-core and multi-threaded machines, a notorious increase in performance and an efficient use of system resources can be achieved by using this type of processing.

It is also possible to use an external control entity with state keeping to isolate tasks. Although not required by this theorem, the action state can be persisted and used to guarantee integrity, resilience and transactional processing. Action states can also be used to resume a previously incomplete workflow execution.

Instability in IS projects

According to NS theory, the penalty for not following these four theorems is the loss of system stability and the occurrence of combinatorial effects. In particular, an IS becomes unstable when:

- a concern is not separated.
- a data entity does not exhibit version transparency.
- an action entity does not exhibit version transparency.
- an action entity is called in a stateless way.

2.4 Software elements

NS theory proposes that data entities and action entities should be encapsulated in software elements that comply with the theorems above. In particular, NS theory proposes the following set of software elements:

Data element

A data element represents a set of data fields, including links to other data elements. Based on the DVT theorem, these data fields should be accessed by methods (such as get and set methods) to achieve version transparency.

Action element

An action element represents a single, encapsulated functional task at a given modular level (SoC theorem). Its arguments and parameters must be encapsulated as data entities, since it needs to comply with the DVT theorem.

An action element needs to support and encapsulate multiple versions of its single functional task, as it needs to comply with the AVT theorem. Therefore, action elements can have multiple implementations, each providing the same functionality in different versions.

Based on SoC and SoS theorems, workflows need to be separated from action entities, and will therefore be encapsulated in workflow elements.

An action element can call another action element or an external system. Therefore, a hierarchy exists of actions elements calling other actions elements.

Workflow element

A workflow element represents a sequence of action elements which are called in a stateful way, by keeping the state at each call (SoS). Workflow elements are concerned with sequences or compositions of action elements. Considering that these actions can invoke other actions or even workflow elements through an action element, this reveals that a hierarchy exists.

Workflow encapsulation, based on SoC theorem, means that workflow elements cannot contain other functional tasks. Based on SoS theorem, workflow elements must be stateful and this state is required in every execution of an action element. Therefore, this action-state needs to be part of, or linked to, the target data element that serves as the action entity target.

Connector element

A connector element represents the structured encapsulation of software constructs into an element that performs an I/O task in an information system. This allows the information system to interact with external systems by manipulating a data element without calling an action element in a stateless way (SoS theorem).

Trigger element

A trigger element represents the structured encapsulation of software constructs into an element that activates a workflow element on a periodic basis, for example. It may also control the states of the system and check whether any action element has to be triggered accordingly. According to the SoS theorem, trigger elements need to control both error and non-error states and check whether some action element has to be triggered in response to those states.

Anticipated changes

In addition to these elements, supporting tasks are added within each structured encapsulation in a way that is consistent with the NS theory theorems and concepts.

NS theory proposes to build information systems based on the aggregation of instances of these five fine-grained high-level software elements [23].

For these elements, there is a set of anticipated changes:

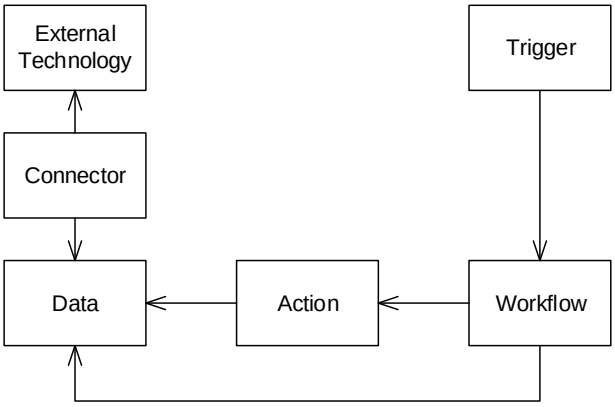


Figure 2.2: Conceptual model of software elements in NS theory

- An additional data field.
- An additional data element.
- An additional action element.
- An additional version of an action element.
- An additional action in a workflow element.
- An additional workflow element.
- An additional connector element.
- An additional trigger element.

It is with respect to these anticipated changes that NS theory guarantees that new versions can be added for all primitives in such a way that the information system remains stable.

2.5 Implementation

The software elements described above, as well as their mutual dependencies, are independent of a specific technology. However, their implementation must take place in some technological environment. The following practices are intended to facilitate the implementation of software elements:

Descriptors

Descriptor files are used to define elements in such way that could be possible to transform platform-independent elements into design patterns expressed on a specific programming language through pattern expanders.

Expanders

Pattern expanders are used to translate the elements from their platform independent descriptor files to the target technology environment using code generation tools. Pattern Expansion is, essentially, automatic code generation of the elements in an information system. All generated artifacts must comply with the NS theorems. After pattern expansion, the source code of the information system is created automatically, but when there is the need to customize the elements, the developer can add custom code within the elements using anchor points that are appropriately marked as such in the generated source code files.

Harvesting

Additionally to the pattern expansion done by the expanders, it could be the need to customize and extend some functionalities of the IS. These customizations introduces context-specific functionalities through manual programming either within anchor points in the expanded files (called “injections”) or in separate files (called “extensions”).

This operation called *harvesting* can be used by the developer to retrieve the previously injected custom code in the elements back to the element descriptors, so that at a later time if the developer regenerates the application, all custom code is automatically re-inserted at each regeneration. This operation is initiated manually by the developer and will occur prior to compile time. Without harvesting, some of the user injected custom code would be lost on a new pattern expansion.

Technological Platform

The current technological platform for NS is based on a 4-Tier Web application architecture implemented with J2EE, using the Jonas Web application server for both the presentation and application logic, providing the connection with the Web browser on the client side, and the relational database at the server side. The four tiers correspond the user interface on the Web browser, the front-end Web application server, Jonas Java Web application server, and the back-end relational database server.

The Enterprise JavaBeans (EJB) framework [24] is used as a server-side component model, persistence is implemented with *entity beans*, which provide the interface to any relational database engine for which a JDBC driver exists. Processing or operations are implemented in *session beans* contained in the same EJB container. Independent threads or engines driving certain tasks are implemented in separate Jonas services.

2.6 Summary

The NS theory is stable by using its five fine-grained software elements to build a software architecture for an IS software application. The determination of the anticipated changes for this elements create a ruleset of what can be done or not, to maintain the desired system stability and evolvability. By using

the element expansion operation in conjunction with the harvesting operation, the developer naturally transforms a descriptive language expressed in platform-agnostic descriptor files into a stable, correctly architectural designed full-fledged IS application, without lose focus from its application main functionality.

The base concepts involved in the NS theory are:

- *Systems theoretic stability*: where a limited input set of a function results in a limited set of output values even for an infinite time;
- *Assumption of unlimited system evolution*: Considering an unlimited period, and an unlimited evolution of the system. The primitives and the number of dependencies between them will indeed become infinite or unbounded for an infinite time.

Both of these concepts must be considered together in order to maintain system stability, for that the impact of a change must not depend of the size of the system.

A definition for NS can be as IS's that are stable with respect to a defined set of anticipated changes where a bounded set of these changes results in a bounded set of impacts to the IS application project.

Combinatorial effects are (hidden) coupling or dependencies, increasing with the size of the system. Combinatorial effects are omnipresent, and a way to circumvent them is the systematic application of these theorems at the IS design:

- *Separation of concerns*: Each concern should be separated from others, considering that they are expected to evolve differently over time. In NS an action entity can only contain a single task (single change driver).
- *Data Version Transparency*: Data entities that are received as input or produced as output by action entities need to exhibit version transparency.
- *Action Version Transparency*: Action entities that are called by other action entities need to exhibit version transparency.
- *Separation of States*: The calling of an action entity by another action entity needs to exhibit state keeping.

These are the foundational concepts present in the NS theory, and all of them will be followed in this work NS architecture proposal. On the next chapter, will be presented some additional concepts and also the proposed architecture.

Chapter 3

Proposed architecture

In essence, the Normalized Systems (NS) theory is intended to define an Information System (IS) architecture that achieves high evolvability in the lifetime of the system in such a way that the cost and ease of changing it remain constant over time, regardless of how many modifications were made in the past [25]. NS theory suggests that high evolvability can be realized by designing a modular structure based on fine-grained software elements and theorems described in the previous section. In this section, it is introduced some additional principles that will allow us to look at the foundations of NS theory in a new perspective that led us to a new architecture proposal.

3.1 Duality principles

From the five software elements defined in the NS theory, one of them is the key element responsible for the process coordination. This element is the Workflow element, which allows the embedding of *process logic*: a set of assumptions, principles, and interrelationships that underlie a business process design and determines its sequence of activities or events. Traditionally, process logic in NS theory is specified in the Workflow element as a state machine that characterizes a sequencing of Actions, and where the execution of each Action brings the process from a previous state into a new state.

By using a target Data element to keep state, the Workflow element possesses an elegant solution to restrict the use of data according to the Separation of Concerns (SoC) theorem and persist the state between each Action element execution. However, this approach confines a business process to adopt only one Data element and also allows the access to the Workflow state within Action elements, injecting process logic details into an element that belongs to *application logic*: software entities that perform a particular atomic task.

An action element represents an atomic task at a given modular level. The IS has to be aware of its starting, ending, and result, but not of its inner workings or progress. Internally, a task could perform calls to other action elements or systems (internal or external). Therefore, a hierarchy exists of actions elements calling other actions elements. This hierarchy is equivalent to a coordination of atomic tasks inside another task creating coupling between them. This hierarchy also relates to the application

behavior. The behavior of an application should be considered to be an independent change driver that can evolve differently from the change drivers found in action entities. Therefore, considering the SoC theorem, *the behavior of an application must be separated from the underlying action elements.*

Usually, the functionality of an information system can be split into *process logic* and *application logic* [26]:

- *Process logic*: is usually implemented and managed through a Business Process Management System (BPMS) either internal or external to the system and it comprises high-level software entities closely related to business processes. Process logic is expressed in the form of a Business Process Modeling Language (BPML) describing all the relations and coordination of application logic components. Its main concern is the application logic components coordination in order to form an higher-level business process.

- *Application logic*: is implemented through the underlying software entities that constitute the application structural components. These components are software entities that performs a particular atomic functional task that are coordinated by process logic.

Duality of Structure and Behavior

In the late 1970s, Data Flow Diagrams (DFDs) [27, 28] became a popular tool for system analysis and design. Basically, DFDs described the data flows across system functions; in the Yourdon/DeMarco notation [27], functions were represented as circles and data flows were represented as arrows between circles. For this technique to be effective, it required the system to be properly divided into a set of functions; therefore, having a correct *functional decomposition* of the system was as important as having a correct specification of the data flows. In other words, the structural and behavioral perspectives of the system were developed in tandem.

Later developments in object-oriented modeling and design [29] have seen such functional decomposition being refined into class diagrams, which brought a significant advance in the structural perspective. On the other hand, the behavioral perspective was being developed more in the realm of process modeling and workflow management [30], although without settling on a standard language, as each workflow system used its own modeling language. The development of Unified Modeling Language (UML) [31] brought the structural and behavioral perspectives under a common umbrella, but in separate diagrams; and the efforts to define a standard process modeling language eventually led to Business Process Model and Notation (BPMN) [32], which has some similarities to UML Activity Diagrams, but has been growing to address specific needs related to the modeling of business processes.

In NS theory, the need for process logic has been addressed by defining an action entity designated as workflow element. The workflow element encapsulates the application behaviour and represents an high-level functional task constituted by a graph-based implementation in the form of a state machine.

Some problems arise from this approach:

- The existence of a hierarchy of action entities creates a source of coupling between actions and

usually higher coupling means higher dependency between software entities, therefore, the probability of having combinatorial effects will rise.

- The lack of a behavioral entity on the software entities model means that behavior is not being addressed on the same level as structure.

Therefore, software entities should be separated between structure and behaviour. The software entities associated with process logic must be differentiated from the software entities associated with application logic. In other words, application logic contains structural software entities, to be coordinated by the process logic which contains behavioral software entities.

The point to retain here is that the need for functional decomposition (structural perspective) and process modeling (behavioral perspective) have always been present in system analysis and design. One cannot go without the other, and if in NS theory the structural perspective is addressed with Data and Action elements for functional decomposition, the same cannot be said about the behavioral perspective, where it lacks an equivalent set of fine-grained elements. Later in this chapter, it is introduced Events and Rules in the behavioral perspective, as the dual elements of Data and Actions in the structural perspective.

The criterias to decide whether an element is a structural or behavioral software entity, are:

- When a software entity that participates in a task characterized as a *unit of work*, defined as an atomic task, is defined to be a structural software entity;
- When a software entity is related with the *coordination* of software entities that relates to one or multiple tasks, is defined to be a behavioral software entity;

A Unit of Work is a widely used concept in software development. [33] This concept states that a task must keep track of everything that had being done during its transaction, and when it finishes, it has to figure out what needs to be done to commit/rollback the results of its work. In NS theory is also used and is known as a task, that is directly related to a single change driver.

Duality of States and Events

In the original NS theory [5], processes are modeled as a state machine where the transition between states takes place through the execution of Actions. Therefore, a state corresponds to the moment in time when the execution of an Action has just finished but the execution of the following Action has not yet begun. In this context, the execution of a Action plays the same role as an event that causes a transition from one state to another. There is one such event between any two consecutive states, and there is always a state between two consecutive events. This mutual relationship between states and events is illustrated in Figure 3.1.

The duality between states and events also exists in other process modeling languages. For example, in Petri nets [34] there are *places* and *transitions* which play a similar role to the states and events depicted in Figure 3.1 (even the notation is similar, since places are drawn as circles and transitions are usually drawn as thick bars or rectangles).

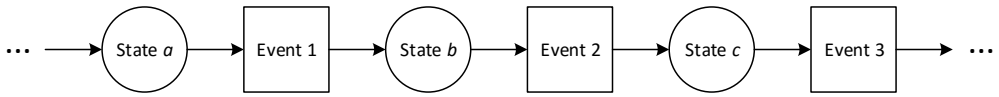


Figure 3.1: States and events

This means that whatever behavior that is specified as a set of states can also be described by focusing instead on the events between those states. The two descriptions are equivalent, and this duality has led to two different approaches to process modeling, namely *graph-based* approaches that are based on states and activities, and *rule-based* approaches that are based on events and rule processing. A survey of these modeling approaches can be found in [26].

For the purpose of modeling process behavior in normalized systems, the difference between the graph-based approach and the rule-based approach can become significant. The graph-based approach favors the definition of the process as a whole, leading to a coarse-grained construct (e.g. a workflow engine, or the Workflow element) that controls process execution from end to end. On the other hand, the rule-based approach favors a decentralized, piece-wise definition of the process, where the behavior emerges from the collective effect of the application of several rules. These rules (and their firing events) are fine-grained constructs that can be combined in several ways to implement a variety of workflow patterns.

In rule-based workflows, rules trigger actions according to their definition and each action invocation is preceded and followed by events. In NS graph-based workflows, a state will trigger an action, or actions, according to the workflow definition. Considering both approaches, it can be concluded that rules triggering actions in a rule-based workflow corresponds to states triggering actions in graph-based workflows.

Therefore, in rule-based workflows, states and events are complementary and between each pair of successive events there is a state, and between each pair of successive states there is an event.

3.2 Graph-based and rule-based workflows

The workflow element in NS theory can be regarded as a behavioral entity, with an implementation based on a graph-based approach by using state machines. The main reason for this choice was its simplicity and close relation with business process modeling languages. Another possible view of the Separation of States (SoS) theorem consists in having a rule-based workflow system where rules trigger actions in the same way as states trigger actions. Therefore, rule-based systems can also be considered to be used as a behavioral software entity.

In a graph-based workflow approach, the process definition contains a set of activities represented as nodes, and a set of control-flow and data dependencies between activities that are represented as arcs between nodes. These graph-based process models provide an explicit specification for the needed process logic. Graph-based approaches for process modeling are usually based on Petri nets theory. The strengths of Petri net based modeling include formal semantics, graphical nature, and abundance

of analysis techniques.

In contrast, in a rule-based workflow approach, process logic is abstracted into a set of rules, each one associated with one or more business activities. A rule-based workflow uses a rule inference engine capable of reacting to events and triggering further actions. This engine examines data and control conditions to determine, at run-time, the best matching action for executing relevant business activities according to pre-defined rules. Rule-based approaches can also be based on Petri nets theory, but with a subtle difference, the resulting net will be conditioned at runtime by the rules conditions.

A comparison between graph-based and rule-based approaches can be found in [26], and it can be summarized as follows:

- *Expressivity*:
  - Both are able to express structure, data, and execution requirements, but temporal requirements are only expressed on the rule-based approach.
  - The graph-based approach can represent the majority of worflow patterns, but a rule-based workflow can represent more workflow patterns than a graph-based workflow.
- *Flexibility*:
  - On the graph-based approach the process model must be completely defined before processes can be executed, and all possible scenarios are explicitly specified.
  - On the rule-based approach the process modeling becomes more flexible as an open specification is supported without the need of a closed and explicit specification.
- *Adaptability*:
  - On the graph-based approach exceptions rise if some behavior occurs that has not been defined in the process model. Therefore, exception handling need to be defined through an additional set of policies.
  - On the rule-based approach expected exceptions can be handled by specific rules.
- *Dynamism*:
  - Changes to a graph-based model must preserve the structural dependencies of running instances. Therefore, the supported changes to process model after deployment are limited.
  - Changes to a rule-based model can be made more freely since the process logic, even for running instances, will emerge based on the event that occur at run-time.
- *Complexity*:
  - The graph-based approach has less complexity in model representation and verification, and requires less expertise when modeling.
  - The rule-based models make it harder to predict and visualize run-time behavior, leading to a more complex verification process.

The main conclusion from this comparison between both approaches is that the rule-based approach can become more suitable to express process logic as behavioral entities in NSs. The main reasons for this are:

- Rules are easier to change.
- When rules are changed, there is a lower impact on running process instances.
- Rules inherently support exception handling by defining appropriate events.
- Rules are more flexible in that the complete behavior does not need to be fully specified beforehand.
- Rules can express structure, data, execution and temporal requirements. They are also capable of expressing more workflow patterns than the graph-based approach.
- There is a lower coupling resulting from the separation of process logic and application logic. Events and rules can be seen as behavioral entities, while data and actions are structural entities.

3.3 ECA rules for process modeling

The idea of using Event-Condition-Action (ECA) rules to define and implement process behavior goes back to the 1990s, with pioneering works such as TriGSflow [6], WIDE [7], SWORDIES [1], and EVE [8]. Since then, several authors have proposed and discussed the benefits of using ECA rules for describing and implementing business processes [9, 10, 11]. Among these benefits, there are some characteristics of ECA rules that are widely recognized and that are of special interest in the context of normalized systems:

- ECA rules are inherently flexible in the sense that they can be changed or adapted to new requirements, and it might even be possible to do this without interrupting running processes.
- ECA rules deal with error handling and exceptions in a natural way. Since errors and exceptions are also events, the rules to handle them can be defined in a similar way to any other rule.
- In traditional graph-based approaches, the transition to a new state usually occurs upon completion of an activity. However, with a rule-based approach it is possible for a single activity to be the source of multiple events, with new rules being fired at any point during execution.

A simple ECA rule has the following form:

ON *event* IF *condition* THEN *action*

ON	**E**vent
IF	**C**ondition
DO	**A**ction

Figure 3.2: ECA rule [1]

This means that when a certain event occurs, a condition is evaluated and, if the condition is true, some action is invoked. When the action raises an event, another rule will be fired, creating a chain of actions and events that is the basis for implementing sequential behavior.

Other common workflow patterns, such as those found in [35], can also be implemented. For example, an AND-split (parallelism) can be implemented with two rules that are fired by the same event but execute different actions. As another example, a XOR-split (branching) can be implemented with two rules that react to the same event but the condition of one rule is the logical negation of the other, so that only one action will be performed.

In some application scenarios, it may be useful to extend ECA rules with multiple events, multiple conditions, and/or multiple actions. With multiple events it is possible, for example, to have AND-joins (synchronization), such as having a rule that is fired only after two different events have occurred; with multiple conditions it is possible to have more elaborate expressions to decide whether to execute an action or not; and with multiple actions it is no longer necessary to define multiple rules when several activities must be done in response to the same event/condition.

In the proposed architecture, it will use these extensions in a purely AND-logic, meaning that *every* event must occur and *every* condition must be true in order to execute *every* action; otherwise, no action will be executed at all. An OR-logic can be built by defining multiple, alternative rules (i.e. rules that respond to different events, or rules that respond to the same event but have different conditions). At this point, it should be noted that the firing of rules is non-deterministic, i.e. there is no predefined order for the evaluation of rules that are fired by the same event/condition.

A simple ECA rule can be extended to support an alternative action by using Event–Condition–Action–Alternative Action [1] (ECAA) with the following syntax:

[H] ON *event* IF *condition* THEN *action* ELSE *alternate action*

ON	**E**vent
IF	**C**ondition
Then DO	**A**ction
Else DO	**A**lternative Action

Figure 3.3: ECAA rule [1]

ECAA rules can be considered a helper to the composition of more complex rules, in fact an ECAA

rule is equivalent to the composition of two equal ECA rules with the condition negated in one of them. Therefore, some derivatives of ECA rules, like the ECAA rules, can only be considered if they are equivalent to a composition of simple ECA rules.

For example, EC^nA^n rules can be used to implement rules that supports a process model that needs decision-table-based processing, where these rules accomplish it by allowing the execution of one chosen condition/action between n condition/action possible branches.

ON	**E**vent
CASE	**C**ondition$_1$
Then DO	**A**ction$_1$
	...
CASE	**C**ondition$_n$
Then DO	**A**ction$_n$
CASE ELSE	**A**lternative Action

Figure 3.4: EC^nA^n rule [1]

EA rule is another type of rule that can be considered, where it is an equivalent rule to an ECA rule containing a condition with a true fixed value.

ON	**E**vent
DO	**A**ction

Figure 3.5: EA rule [1]

Some constructs are needed to implement process and workflow modeling through ECA rules. Examples are:

- A sequence of actions that can be achieved by simply sequencing events and actions together.

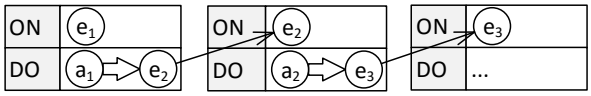


Figure 3.6: Modeling rules to achieve sequential actions [1]

- Parallel actions that can be achieved by matching multiple rules with one event and later joining them by matching all the events from each branch in one rule.

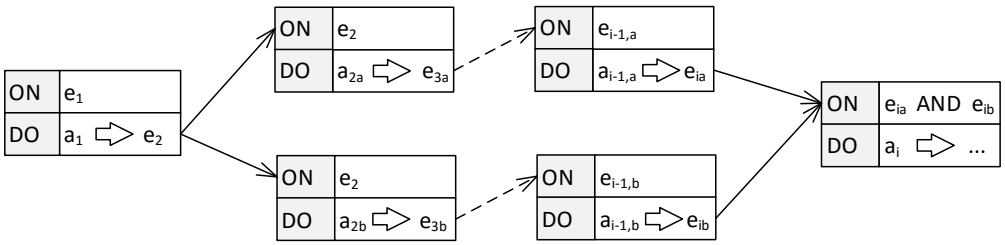


Figure 3.7: Modeling rules to achieve parallel actions [1]

- Alternative actions that can be achieve by using an ECAA rule. At the beginning, a path is chosen by the condition of this first rule; then in the end of each branch a last rule join the execution flow by using an OR operation between the completion events of each branch.

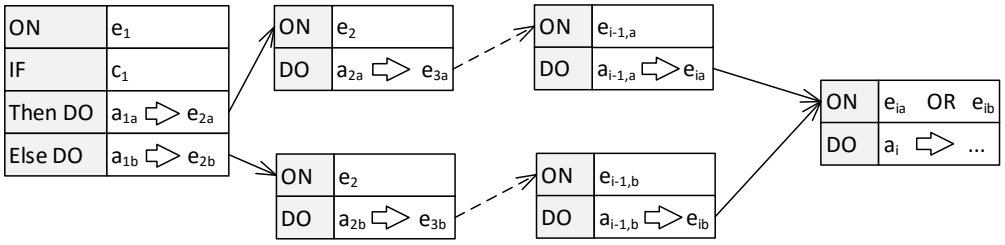


Figure 3.8: Modeling rules to achieve alternate action branches [1]

- Iterations of actions can be achieved by using an ECAA rule to decide if the execution continues or jumps back to an already evaluated rule.

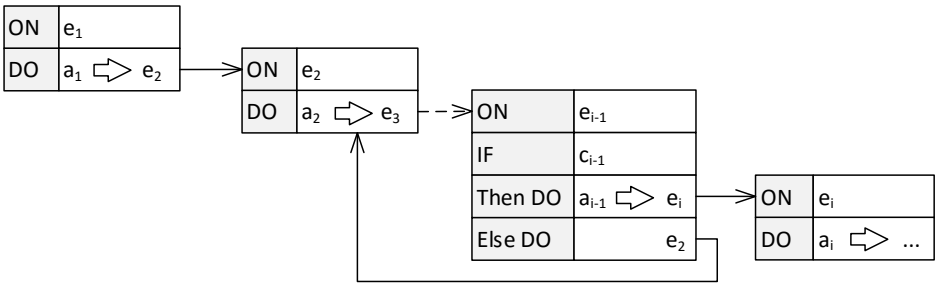


Figure 3.9: Modeling rules to achieve iteration of actions [1]

From these examples, it is clear that rule-based workflows only by using simple ECA rules are capable of expressing a large set of workflow constructs. Rule-based workflows tend to have higher complexity than graph-based but with the substitution of ECA simple rules with helpers like the ECAA rule, they can be simplified and even become simpler than their counterpart.

3.4 Run-time behavior with ECA rules

With the adoption of events and rules in this work NS architecture, some aspects and concepts will be present. It is essential to analyze each one and how can they be correct in the light of NS theory concepts:

Rules are concurrent

In a rule-based approach, it is possible to have a parallel execution flow where several concurrent actions are triggered by the raise of a single event. Therefore, rules are concurrent and can be triggered in a non-deterministic way. There should be some degree of isolation between actions to prevent potential problems that may arise from such concurrency, namely concurrent access to data, concurrent invocations of the same resources (SoS theorem). This introduces the need for the next principle as well.

Action isolation

NS theory already defines that one action can only have one task, and this task is directly related to one independent single change driver, denoting some action atomicity (SoC theorem). In fact, considering that actions can be invoked in a concurrently non-deterministic way with other actions, a conflict can occur when multiple actions are sharing the same data entities, leaving these data entities in a non-deterministic state. Therefore, it must be assured that each Action is isolated from other Actions (SoS theorem) where the execution of an Action cannot never restrict or influence another Action.

Process correlation

Considering the duality of states and events, the sequence of states and the succession of events brings the need to identify each sequence with an unique identifier. Also by considering that rules can lead to parallel execution of actions, each branch must be identified by this unique identifier already defined for the parent branch. This concept is especially useful to be able to establish a correlation between a business process instance and its constituent actions. The multiple executing process flows of an IS application must be isolated from each other and tagged using their own correlation id.

The use of correlation ids is an import mechanism in service interactions [36], where it is often necessary to associate an incoming message with a specific service instance. This mechanism is used to associate an incoming Event with a Rule instance. This allows multiple instances of the same business process to be active at the same time, since the Events and Rules from one process instance are managed separately from the Events and Rules of other process instances (SoS theorem).

Events are non-deterministic

The raise of an Event element by an Action element is considered to be non-deterministic, using other words, it will not wait for an expected response. Therefore, when an Action raises an Event, it will not make any assumption, and it will not hold any dependency from the subsequent execution. By the SoS theorem, an Action element cannot invoke another Action without keeping state and by using a rule-based approach this state is changed every time an Event is raised, that by consequence can trigger Rules. Therefore, this state keeping is ensured by the event-rule processing. An Event has to symbolize a fact related to the Action from where it is raised and not a particular Action that needs to be called from that point. By the SoC theorem, an Action can only be concerned with its internal functional task and with what is directly related to it. Therefore, when an Event is raised, it should be only concerned with what happened inside the scope of that Action task and nothing else.

In conclusion, it is easy to recognize that the use of events and rules strengthens the presence of SoCand SoS theorems in this works architecture. The atomicity of Actions is also strengthened where actions become completely isolated from each other and the behavior elements.

3.5 Architecture

Previously on chapter 1 it was stated that some of the fundamental principles on which NS theory is built, will be used as guidelines for the design of the proposed architecture, namely:

- the principle of having a *fine-grained modular structure* which can accommodate changes without requiring an extensive system overhaul;

- the principle of having *low coupling* and *high cohesion* between elements so that a change in one element has the least potential to generate impact on other elements.

All elements will inherit the characteristics of the base Element (Figure 3.10). This base Element has a unique name across the entire application. Additionally, it also has an unsigned integer version number to identify each version of the Element, and for each version of an Element the architecture has a separate instance having their relations managed by their version transparency implementation. To be correctly identified from the rest of the elements inside an application each Element will use the tuple of their name and version. Every time an Element requests a lower version reference to another Element, a Liskov Substitution Principle (LSP) [37] compatible conversion function must be applied to downgrade the Element in a correct way.

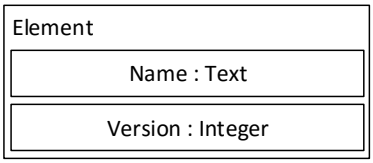


Figure 3.10: Base element conceptual diagram

Field element

The achievement of a fine-granularity is the main motivation for the introduction of the Field element in the architecture, mainly because each field inside a Data element can be considered an independent change driver. Therefore, they must be separated from their container (Data element) and from each other (SoC theorem).

The Field element, represented on figure 3.11, has the main concern of holding a single primitive value. The possible primitive types to use on this element are defined in appendix (Table B.1).

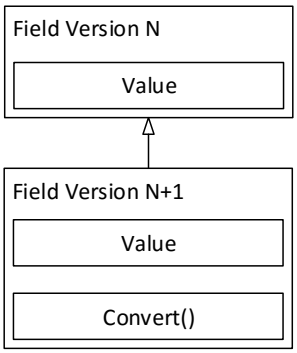


Figure 3.11: Field element conceptual diagram

Anticipated changes

On this element, it is only possible to change the data type of its contained primitive. Therefore, the anticipated change for this element is the change of its data type, but because it is not an addition type of change it can only be valid if it is correct with the Liskov Substitution Principle (LSP) through a conversion function.

Customizations

The only suitable customization for this element is the definition of a conversion function to transform the actual data type into the previous data type.

Data element

The Data element, represented on figure 3.12, assumes the main concern of containing a set of references to Field element instances. Each Field item is a reference to a particular Field element version. Each Field type is unique inside a Data element, and each one would be accessible by a public property (equivalent to get/set methods in other programming languages) named with the same name of the Field element.

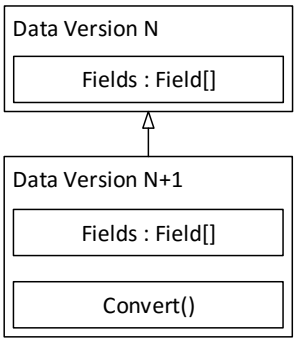


Figure 3.12: Data element conceptual diagram

Anticipated changes

The anticipated changes for this element are:

- Addition of a Field element into the Data element;
- Removal of a Field element from the Data element;
- A contained Field element reference version change;

Customizations

Similarly to the Field element, the only suitable customization for this element is the definition of a conversion function to transform the actual data element version into a previous data element version (LSP compatible).

Event element

The Event element, presented on figure 3.13, represents a fact or an intent related with the IS. An Event can be raised in Action elements representing a fact only related to that particular Action (SoC theorem).

When an Action raises an Event, it will no expect to receive any result from this operation (Chapter 3.4). An Event can also be raised from outside the IS application by an external system and in this case, the Event will represent an external system fact or an intent related to the IS application (SoC theorem). All Events when are raised they will be cloned when thaey are set in Rules during the rule

engine processing. Therefore, providing isolation from the system that raised it and from all the multiple Rules that can receive the raised Event. Additionally, it also delivers the ability to reuse or modify the Event right after a raise of an Event (SoS theorem).

Each Event instance have a unique identifier property representing the correlation id of the process where it is being handled. The correlation id on each Event starts with a empty value being set afterward in the Rule engine processing. Each correlation id represents a unique process.

The Event element assumes the main concern of representing a fact or intent, but these facts or intents could need to reference Data elements to support them. Therefore, the Event element contains a set of references to Data element instances. Each Data item is a reference to a particular Data element version. Each Data type is unique inside an Event element, and each one would be accessible by a public property named with the same name of the Data element.

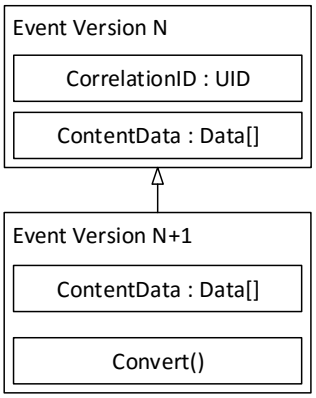


Figure 3.13: Event element conceptual diagram

Anticipated changes

The anticipated changes for this Element are:

- Addition of a Data element into the Event element;
- Removal of a Data element from the Event element;
- A contained Data element reference version change;

Customizations

The only suitable customization for this element is the definition of a conversion function to transform the actual Event element version into a previous Event element version (LSP compatible).

Action element

The Action element, presented on figure 3.14, represents an atomic and isolated task closely related to a single change driver (SoC theorem). An Action can manipulate Data elements present in the InputData

set. An Action can also manipulate and raise Event elements present in the OutputEvents set. Each Data element in the InputData set and Event element in the OutputEvents set must be unique with respect to their type.

Each instance of an Action element must be isolated from others Action elements (SoS theorem). Every time a Data element is transferred as an argument to an Action element it is cloned guaranteeing the isolation of the Action from the previous manipulative Actions (SoS theorem). Therefore, each time an Action receives a Data element, in fact, it is receiving a snapshot of that Data element. Like referred in the Event element description, each time an Event element is raised in an Action, it will be cloned before being processed by the Rule engine, guaranteeing the isolation of the actual Action to the next manipulative Actions (SoS theorem).

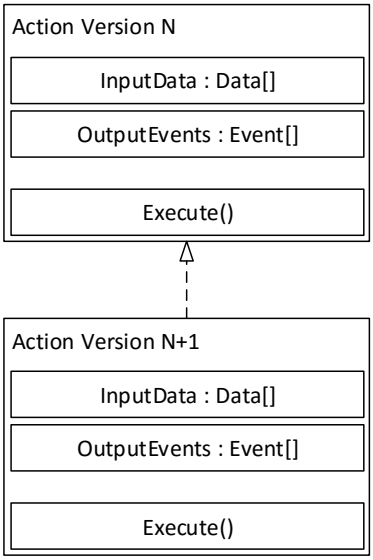


Figure 3.14: Action element conceptual diagram

Anticipated changes

The anticipated changes for this Element are:

- Addition of a Data element into the InputData set;
- Removal of a Data element from the InputData set;
- A contained Data element reference version change;
- Addition of a Event element into the OutputEvents set;
- Removal of a Event element from the OutputEvents set;
- A contained Event element reference version change;
- An execute function modification;

Customizations

The customization for this element is the definition of the Execute function to develop the internal task definition on the behalf of the developer.

Condition element

The Condition element, presented on figure 3.15, has the main concern of checking the Event elements of a Rule element and vote yes or no to the execution of the Rule element referenced Action elements. Once again, like the Field element, the Condition element represents an independent change driver on its own. Therefore, it must be separated from its calling Rules (SoC theorem).

The Condition element assumes the main concern of checking the Event elements of a Rule element, but to perform this check it needs to reference the Event elements of the Rule element. Therefore, the Condition element contains a set of references to Event element instances. Each Event item is a reference to a particular Event element version. Each Event type is unique inside a Condition element, and each one will be accessible by a public property named with the same name of the Event element.

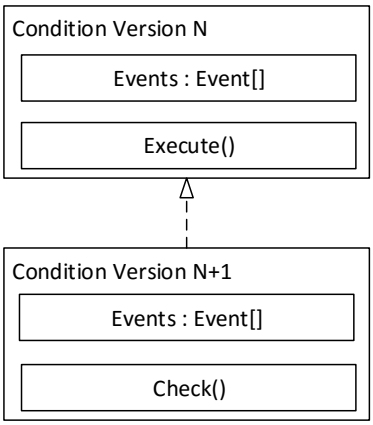


Figure 3.15: Condition element conceptual diagram

Anticipated changes

The anticipated changes for this Element are:

- Addition of a Event element into the Events set;
- Removal of a Event element from the Events set;
- A contained Event element reference version change;
- A check function modification;

Customizations

The customization for this element is the definition of the Check function to develop on the behalf of the developer.

Rule element

The Rule element, presented on figure 3.16, is the “glue” of the entire architecture. Its main concern is to listen for specific Event elements raised in the system, verify them using its Condition elements and then invoke its Action elements. It contains sets to its listening Event elements, checking Condition elements and invokable Action elements. Each one of this items is a reference to a particular Element version. Each Element type is unique inside a Rule element, and each one would be accessible by a public property named with the same name of the Element.

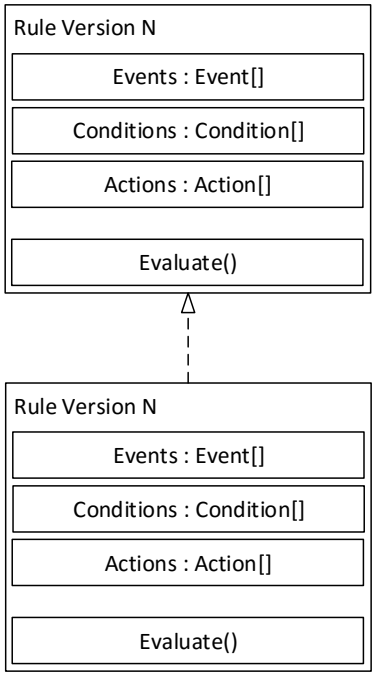


Figure 3.16: Rule element conceptual diagram

Anticipated changes

The anticipated changes for this Element are:

- Addition of a Event element into the Events set;
- Removal of a Event element from the Events set;
- A contained Event element reference version change;
- Addition of a Condition element into the Conditions set;

- Removal of a Condition element from the Conditions set;
- A contained Condition element reference version change;
- Addition of a Action element into the Actions set;
- Removal of a Action element from the Actions set;
- A contained Action element reference version change;

Customizations

There is no suitable customizations for this element. Everything will be defined on the Element expansion of in the base framework.

Application

The Application object is introduced with the objective of aggregating the fine-granular IS software elements into a single envelope. The interaction from external systems are ensured by the Application element, where an external system can raise Events in the system. Additionally, the rule engine processing of events is also implemented internally in this Application object.

Figure 3.17 illustrates the proposed architecture, which comprises a structural perspective and a behavioral perspective. The structural perspective contains the standard Data and Action elements from NS theory, which provide a fine-grained set of elements for the functional decomposition of a system. On the other hand, the behavioral perspective introduces the new Event and Rule elements to provide a fine-grained set of elements for the implementation of process behavior.

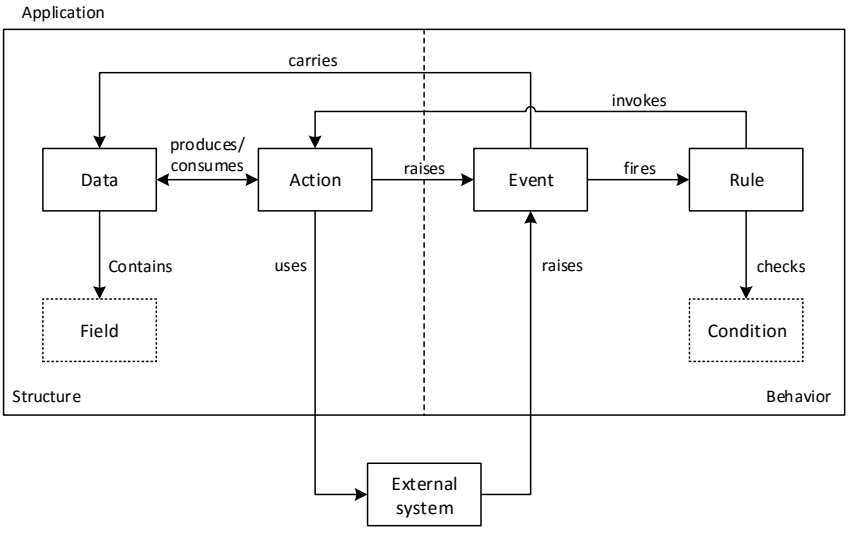


Figure 3.17: Overview of the proposed architecture

The original Connector and Trigger elements are not included in the architecture. The original Connector element supports the interaction between an external system and a Data element. By adopting

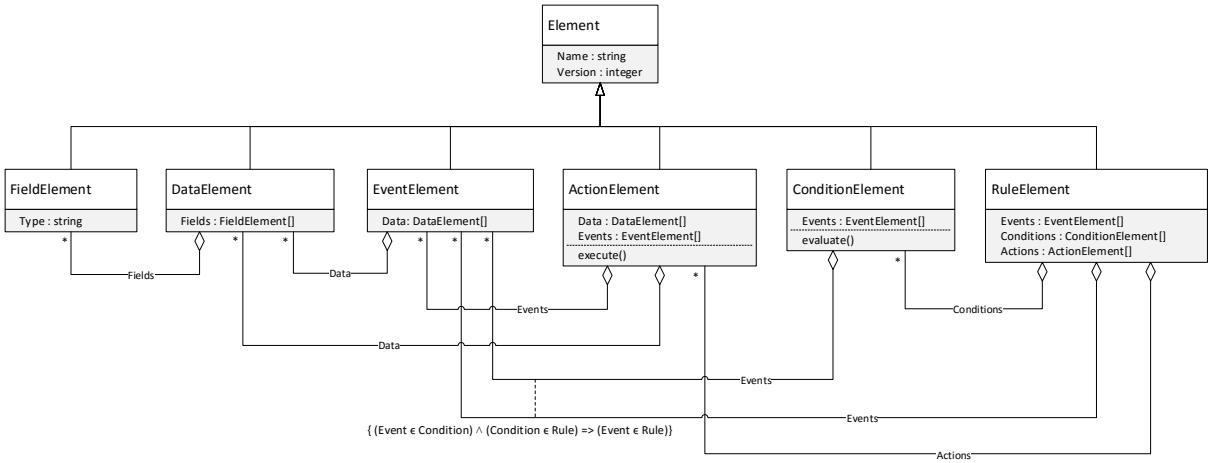


Figure 3.18: Class diagram with inheritance and aggregation relationships

Rules and Events, this interaction is more suitable to be made through the raise of Events and Rule processing (SoS theorem). In the original architecture, the Trigger element is applied as a service where a Workflow is invoked, triggered periodically or a reaction to a state. By adopting Rules and Events, this type of triggering can be ensured by an external system. This external system can interact with the system through the raise of Events and Rule processing (SoS theorem).

Elements interaction

The way the architecture software elements interact with each other can be described as follows (Figure 3.18):

- *Application aggregates software elements*: The Application object acts as an aggregation of all software elements of an IS software application.
- *Data contains Fields*: Data elements acts as an envelope to Field elements. Field elements are a fine-grained (SoC theorem) element that holds a value of a primitive type.
- *Actions manipulate Data*: Data elements can be both consumed and produced by Actions, as in the original NS theory. However, here it is introduced another kind of output from Actions: Events.
- *Actions raise Events*: The most common scenario is for an Action to raise an Event upon completion. However, it is also possible for an Action to generate Events at any point during its execution. Once an Event is raised, it is cloned and processed by the rule engine leaving the original Event isolated from the following processing (SoS theorem).
- *Events carry Data*: An Event has a payload comprising zero, one, or more Data elements. Such Data are inserted in the Event when it is created. Events can be accessed with the same interface of its constituent Data elements (similar to a union in C).
- *Events fire Rules*: Once an event is raised, there may be zero, one, or more Rules that are listening for that Event. All such rules will be evaluated concurrently, in non-deterministic order (SoS

theorem).

- *Rules have Conditions*: Conditions are logical expressions over the content of Data elements carried by Events. If a Rule has multiple conditions, all those expressions must evaluate to true for the Rule to execute its Actions.

- *Rules invoke Actions*: A Rule may execute a set of one or more Actions. Execution is asynchronous, meaning that the Rule invokes each Action and returns immediately without waiting for the Action to complete (SoS theorem).

- *Rules pass Data as input to Actions*: When a Rule invokes an Action, it may pass zero, one, or more Data elements as input. Each of these Data elements must have been brought to the Rule by one of its firing Events. Therefore, a mapping will occur. In this mapping, these Data elements are cloned leaving the original Data element isolated from the following processing (SoS theorem).

- *External systems raise Events*: The way that an external system interact with the IS, is through the Application object where it raises Events into the system starting the followed processing (SoS theorem).

- *Actions uses External systems*: The way that the system can interact with an external system, is by call it from the inside of Actions. This calling must comply with the SoC theorem where an Action must have only one change driver, therefore, by interacting with an external system can only perform this interaction.

Versioning

One of the fundamental principles of NS theory is version transparency [38], and from the description made for the architecture, it may appear that is supported both the addition and removal of elements, while the original NS theory supports only additions and treats removals as a matter of garbage collection [19]. Supporting only additions of elements is the most basic way to support version transparency because it will be natural to suppose that an element not known in the system could not produce any combinatorial effect because it will not exist any reference to it. In a real-life scenario, it will be necessary to perform different types of changes additionally to the addition of elements. The modification of a field data type is a common change. The suppression of a field from a version of a data element it is also a current change.

Similarly to the versioning in the original NS theory, the multiple versions of an element will be packed together. When an element requests a reference to a specific version of another element, it could be necessary to apply a transformation to convert an element from version $n + k$ into an element of version n. It is relevant to refer that this is only valid for backward compatible conversions.

Considering that an element of version $n + k$ is a subtype of an element of version n, there is a principle in object-oriented programming that defines what rules must be applied in the versioning of elements for them be considered to be correct and stable. This principle is the Liskov Substitution

Principle (LSP) [37]. LSP states that if S is a subtype of T then objects of type S must be able to replace objects of type T without altering any of the properties of this type (e.g., correctness, functional task).

When an element ot element version is no longer referenced, it can be garbage collected, as advocated by standard NS theory.

3.6 Rule engine

An essential part of that common functionality is the *rule engine*, i.e. the component that manages the execution of the application at run-time. This rule engine is based on an event queue and a rule instantiation mechanism, as depicted in Figure 3.19.

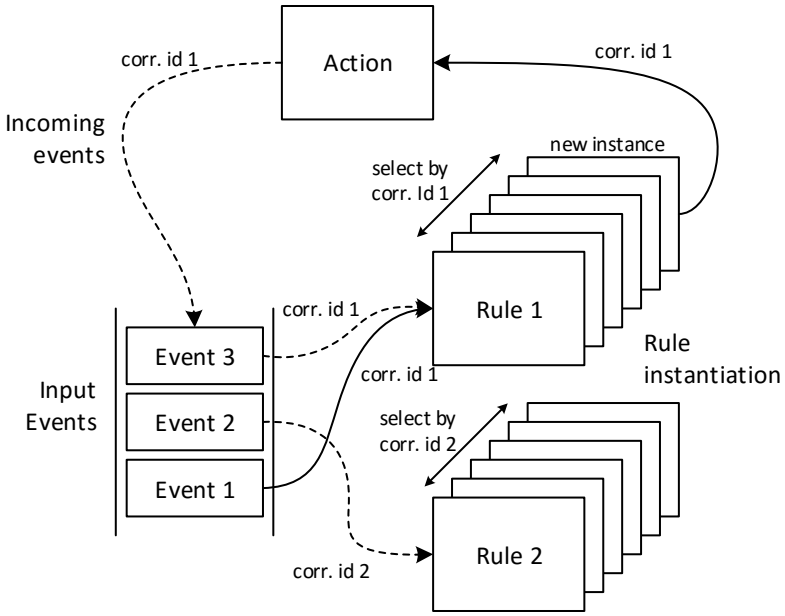


Figure 3.19: Rule engine

First, it is considered the scenario where each Rule is fired by a single Event. In this case, the rule engine receives an Event from the system or from an external system and searches for Rules that are waiting for such Event. It then instantiates such Rules, set their Events, evaluates their Conditions and if the Conditions yield true their Actions are invoked. In turn, these Actions will generate new Events that will be added to the queue. In the meantime, the previous Rule instances can be discarded.

For Rules that are waiting for multiple Events, the Rule is instantiated upon the occurrence of the first Event, and such instance is left alive, waiting for the remaining events to arrive (no Conditions are evaluated at this point because Conditions are only evaluated when all Events are present). The next Event to arrive must have the same *correlation id* as the first Event in order to be associated with that particular Rule instance. Once all of the required Events with the same correlation id have arrived, the Rule instance is fired, i.e. its Conditions are evaluated and its Actions are invoked.

When a Rule instance invokes an Action, it passes its own correlation id to the Action. In turn, the Action will insert this correlation id into any Events that it generates during its execution. This same

correlation id will be used as the correlation id for any new Rule instances that are created from such Events. In general, Rule instances are selected by the correlation id of the incoming Event. If no Rule instance with such correlation id exists, a new one is created.

3.7 Summary

The duality of structure and behavior principle proposes that the functionality of an Information System (IS) should be split into *process logic* and *application logic*:

- *Application logic*: Implemented through the underlying software entities that constitute the application structural components. These components are software entities that performs a particular atomic functional task that are coordinated by process logic.
- *Process logic*: Its main concern is the application logic components coordination in order to form an higher-level business process.

Therefore, software entities should be differentiated between structure and behavior. Software entities associated with process logic must be distinguished from the software entities associated with application logic. In other words, application logic contains structural software entities, to be coordinated by the process logic which contains behavioral software entities.

- When a software entity that participates in a task characterized as a unit of work, defined as an atomic task, is defined to be a *structural software entity*;
- When a software entity is related with the coordination of software entities that relates to one or multiple tasks, is defined to be a *behavioral software entity*;

The duality of states and events proposes that states and events are complementary and between each pair of successive events there is a state, and between each pair of successive states there is an event.

A graph-based workflow approach (like in the original NS architecture) favors the definition of the process as a whole, leading to a coarse-grained construct that controls process execution from end to end. On the other hand, a rule-based workflow approach favors a decentralized, piece-wise definition of the process, where the behavior emerges from the collective effect of the application of several rules. These rules (and their firing events) are fine-grained constructs that can be combined in several ways to implement a variety of workflow patterns.

In rule-based workflows, rules trigger actions according to their definition and each action invocation is preceded and followed by events.

The idea of using Event-Condition-Action (ECA) rules to define and implement process behavior is well known and since the 90s. There are some characteristics of ECA rules that are widely recognized and that are of special interest in the context of normalized systems:

- ECA rules are inherently flexible in the sense that they can be changed or adapted to new requirements.
- ECA rules deal with error handling and exceptions in a natural way.
- ECA rules are a piece-wise definition of a process.
- However, hard to control and understand after a certain complexity.

Some aspects and concepts arises with the adoption of events and rules and it is essential to refer each one:

- *Rules are concurrent*: It is possible to have a parallel execution flow where several concurrent actions were triggered by the raise of a single event, each one raising new events. Therefore, rules are concurrent and can be triggered in a non-deterministic way.
- *Action isolation*: Actions can be invoked in a concurrently non-deterministic way with other actions, a conflict can occur when multiple actions are sharing the same data entities. Therefore, the execution of an action cannot never restrict or influence another action.
- *Process correlation*: The sequence of states and the succession of events brings the need to identify each sequence with an unique identifier.
- *Events are non-deterministic*: The raise of an Event element by an action is considered to be non-deterministic, using other words, it will not wait for an expected response.

Using this concepts in mind and the original NS theory an software architecture is presented in section 3.5 followed by the conceptual and algorithmical description of the rule processing engine (Section 3.6).

Chapter 4

Implementation

Originally, the development framework for normalized systems was implemented over a Java Platform, Enterprise Edition (J2EE) platform based on Java technology and structured as a 4-Tier Web application architecture [5]. Unfortunately, this reference implementation is not publicly available, making it difficult for the wider community to understand how the concepts of NS theory can be transformed into actual application code. This work approach was to develop a prototype implementation of the proposed architecture using only the available information on Normalized Systems (NS), namely the authors book and published articles, leveraged by the rapid prototyping capabilities of the .NET Framework and Visual Studio (VS).

In particular, the C# programming language, the .NET Framework, and the VS IDE were chosen for the following main reasons:

- the .NET compilers and certain components of the .NET Framework have been recently open-sourced, e.g., the Project “Roslyn” on GitHub;
- it is possible to achieve a tight integration with Visual Studio through its Software Development Kit (SDK) and extensibility tools;
- it provides sophisticated code-generation capabilities through the use of Text Template Transformation Toolkit (T4) templates;
- great capabilities to work with asynchronous code, namely with the “async/await” pattern;
- great capabilities to work with complex iterations and expressions, namely using Language-Integrated Query (LINQ);
- the .NET Framework has a very extensive set of libraries and capabilities that makes it suitable to develop any kind of application.
- in general, this platform and its tools offer great capabilities for Rapid Application Development (RAD).

It is relevant to refer that the base framework code and the expanded code (created by the code-generation expansion tool) will not be restricted to any specific type of application. The only requirement

for the architecture code is the standard core .NET Framework. Therefore, it can be used for all kinds of applications supported by .NET, e.g., Web Services, Web applications, Command Line Interface (CLI) applications, desktop application, and others.

4.1 Implementation of software elements

In this section, it will be described each element implementation in all three layers of the architecture: base library; expanded code; custom code (Section 4.3). This description of each element is based on a simple application created for the purpose of demonstrating the architecture. This application is called "Hello world" and has the task of asking the user name's and print a message with the name contained in it.

Base element implementation

The very beginning of this implementation starts with the definition of the `ElementInfo` class (Listing 4.1). This class is useful to identify each Element object across the entire Application. To perform this concern, the `ElementInfo` has defined properties for the name and version values. The redefinition of the standard C# object methods `GetHashCode()` and `Equals()` is needed to correctly identify each Element inside standard C# `Collections` using the name and version properties values.

```
1  public sealed class ElementInfo
2  {
3     public string Name { get; set; }
4     public uint Version { get; set; }
5
6     public override int GetHashCode()
7     {
8        return Name.GetHashCode() ^ Version.GetHashCode();
9     }
10
11    public override bool Equals(object obj)
12    {
13       if (!(obj is ElementInfo)) return false;
14       var other = (ElementInfo)obj;
15       return Name == other.Name && Version == other.Version;
16    }
17 }
```

Listing 4.1: ElementInfo source (Base code[1])

The Element base class will be inherited by all software elements of the architecture. Its main concern is the definition of an element name and version for each type of software element. It also introduces the responsibility to each inherited software element of defining their element name and version.

```
1  public abstract class Element
2  {
3     public abstract ElementInfo ElementInfo { get; }
4  }
```

Listing 4.2: base Element source (Base code)

[1]The comment inside the parenthesis refers that the listing belongs to the base framework source code, or to custom-developed source code, or to expanded source code created by the code-generation expander tool.

Field element implementation

The first element to implement is the Field element, namely because this element it is the most fine-grained and simpler to realize and understand. The main purpose of this element is to hold a primitive value. Considering this concern, it must hold a value on an underlying object protected property. Additionally to this property, this class (Listing 4.3) also defines overridden versions of the standard C# object methods (`ToString()`, `GetHashCode()`, `Equals` and the equality operator(== and !=)), mostly because this envelope class must reflect these methods results according to its underlying value object.

```
1  public abstract class FieldElement : Element
2  {
3     public object Value { get; set; }
4
5     public override string ToString()
6     {
7        return (Value ?? string.Empty).ToString();
8     }
9
10    public override int GetHashCode()
11    {
12       if (Value == null) return 0;
13       return Value.GetHashCode();
14    }
15
16    public override bool Equals(object obj)
17    {
18       if (ReferenceEquals(null, obj)) return false;
19       if (ReferenceEquals(this, obj)) return true;
20       if (obj.GetType() != this.GetType()) return false;
21       FieldElement other = obj as FieldElement;
22
23       return this == other;
24    }
25
26    public static bool operator ==(FieldElement a, FieldElement b)
27    {
28       if (object.ReferenceEquals(a, b)) return true;
29       if (((object)a == null) || ((object)b == null)) return false;
30       if (object.ReferenceEquals(a.Value, b.Value)) return true;
31       if ((a.Value == null) || (b.Value == null)) return false;
32
33       return a.Value == b.Value;
34    }
35
36    public static bool operator !=(FieldElement a, FieldElement b)
37    {
38       return !(a == b);
39    }
40 }
```

Listing 4.3: Field element source (Base code)

However, this Field element base class must be extended to support specific data types. Below on listing 4.4 is presented the generic class that extends the Field element class through the use of *generics* to add support to the field data types. This newly created Field element class performs this extension by hiding the previously defined value property with a new one that casts the value with the proper C# data type. Additionally, is defined a generic `Convert` method that performs a blind conversion of the underlying value object into another Field element value object, however, this method can be overridden in custom code. An implicit cast operator is also defined (taking advantage of the type `T` defined using generics) with the objective to access the underlying value transparently.

```
1  public abstract class FieldElement<T> : FieldElement
2  {
3     public new T Value
4     {
5        get
6        {
7           return (T)(base.Value ?? default(T));
8        }
9        set
10       {
```

```
11              base.Value = value;
12          }
13      }
14
15      public void Convert(FieldElement dest)
16      {
17          dest.Value = Value;
18      }
19
20      public static implicit operator T(FieldElement<T> field)
21      {
22          return field.Value;
23      }
24  }
```

Listing 4.4: Generic Field element source (Base code)

Below on listing 4.5 there presented an example where the Field element class is extended to support the `Integer` data type. It is also determined the name and version of the Field element class. This listing is a good example of automatically expanded source code.

```
1  public partial class PersonId : FieldElement<int>
2  {
3      public override ElementInfo ElementInfo { get; }
4          = new ElementInfo() { Name = "PersonId", Version = 1 };
5  }
```

Listing 4.5: Int32 Field element example (Expanded code)

Data Version Transparency (DVT) theorem

In the generated source code, there was two hypothesis to implement version transparency in Field elements: one is through *inheritance* (where the new version is a subclass of the previous version) and another is through *composition* (where the new version contains an instance of the previous version). From this two hypothesis, it was chosen the composition approach because inheritance will introduce strong dependencies between the multiple versions of the Field element.

```
1  public partial class PersonIdVersion2 : FieldElement<string>
2  {
3      public override ElementInfo ElementInfo { get; }
4          = new ElementInfo() { Name = "PersonId", Version = 2 };
5
6      private PersonId _base;
7
8      public new string Value
9      {
10         get
11         {
12             return base.Value;
13         }
14         set
15         {
16             base.Value = value;
17             Convert(_base);
18         }
19     }
20
21     public PersonIdVersion2()
22     {
23         _base = new PersonId();
24     }
25
26     public static implicit operator PersonId(PersonIdVersion2 obj)
27     {
28         return obj._base;
29     }
30 }
```

Listing 4.6: Field element version 2 (Expanded code)

Like in the example presented in listing 4.6, the Field element contains a private field for the previous version of the Field element. The `Value` property was hidden with a newly defined property (Listing 4.6

```
1  public partial class PersonIdVersion2
2  {
3     public void Convert(PersonId dest)
4     {
5        if (Value != null)
6        {
7           dest.Value = int.Parse(Value.Replace("PERSON", string.Empty));
8        }
9     }
10 }
```

Listing 4.7: Field element version 2, custom developed `Convert` function (Custom code)

line 8-19) that accesses both the actual value and the previous value. On this access, the actual value is returned when the value is requested. However, when the value is stored it will both store it on the actual and the previous value. The previous value will be first converted through a conversion (`Convert`) method. Additionally, it has an implicit operator that will transform the version $n + 1$ Field element into a n Field element, in this case, it will return the contained `_base` object. In the listing 4.7 it is presented an example of a custom developed `Convert` method, where this method overrides the generic Field element `Convert` function in order to provide an Liskov Substitution Principle (LSP) compatible conversion between Field element versions.

This means that an element of `Version2` can be implicitly cast to an object of `Version1` in all those elements which have not been upgraded yet. This approach is preferred, for the main reason that it introduces fewer dependencies between different versions of same element. In addition, the connection between versions is more explicit in the expanded code. A caveat is that both `Version1` and `Version2` will have to be derived from a common base class (e.g. `FieldElement`), while in the previous approach `Version2` inherited that behavior through `Version1`.

```
1  public partial class PersonIdVersion3 : FieldElement<ulong>
2  {
3     public override ElementInfo ElementInfo { get; } = new ElementInfo() { Name = "PersonId",
          Version = 3 };
4
5     private PersonIdVersion2 _base;
6
7     public new ulong Value
8     {
9        get
10       {
11          return base.Value;
12       }
13       set
14       {
15          base.Value = value;
16          Convert(_base);
17       }
18    }
19
20    public PersonIdVersion3()
21    {
22       _base = new PersonIdVersion2();
23    }
24
25    public static implicit operator PersonId(PersonIdVersion3 obj)
26    {
27       return obj._base;
28    }
29
30    public static implicit operator PersonIdVersion2(PersonIdVersion3 obj)
31    {
32       return obj._base;
33    }
34 }
35
36 public partial class PersonIdVersion3
37 {
38    public void Convert(PersonIdVersion2 dest)
39    {
```

```
40          dest.Value = string.Format("PERSON{0:D8}", Value);
41      }
42  }
```

Listing 4.8: Field element version 3 (Expanded & Custom code)

In the listing 4.8 it is presented an example that defines a third version of the `PersonId` Field element. From this example, it is clear to understand that the interfacing between versions is trivial as the creation of a cascade of implicit operators, all accessing the same `_base` element. With this approach, this cascade of implicit operators will create a stack of calls to the multiple `Convert` methods within the stack of versions. Therefore, respecting the multiple version of a Field element.

Data element implementation

The Data element has the concern of holding a register in application data, however in other words, it has the concern of being an aggregation of Field elements. In listing 4.9 is presented the base class for the Data element, where is defined a `Dictionary` to hold the collection of Field elements. Additionally, there is defined a generic `Convert` function that will iterate through the Fields collection and the Fields collection from another Data element assigning the Fields in common with compatible versions. This assigning takes advantage of the underlying Field elements version transparency when accessed through the expanded properties.

```
1  public abstract class DataElement : Element
2  {
3      public Dictionary<string, FieldElement> Fields { get; }
4          = new Dictionary<string, FieldElement>();
5
6      protected void Convert(DataElement data)
7      {
8          (from forig in Fields.Values
9           join fdest in data.Fields.Values
10          on forig.ElementInfo.Name equals fdest.ElementInfo.Name
11          where forig.ElementInfo.Version >= fdest.ElementInfo.Version
12          select forig.ElementInfo.Name).AsParallel().ForAll(
13              result => data.Fields[result] = Fields[result]);
14     }
15 }
```

Listing 4.9: Data element source (Base code)

In the listing 4.10 is presented an example class for a `Person` class containing an `Id` and a `Name` Field. In the first two lines of the Person Data element class is defined the name and version of the element. By its definition, the Data element class must provide access to each Field element contained in its Fields set. Therefore, a property for each Field element is generated (Listing 4.10 lines 6-14) to provide this access, and in each of these properties, the contained Field element is cast with the correct type of Field. In this cast, it is also guaranteed that the required Field version transparency is performed.

```
1  public partial class Person : DataElement
2  {
3      public override ElementInfo ElementInfo { get; }
4          = new ElementInfo() { Name = "Person", Version = 1 };
5
6      public PersonId PersonId
7      {
8          get
9          {
10             return Fields["PersonId"].Cast<PersonId>();
11         }
12         set
13         {
14             Fields["PersonId"] = value;
15         }
```

```
16      }
17
18      public PersonName PersonName
19      {
20          ...
21      }
22
23      public Person()
24      {
25          Fields["PersonId"] = new PersonId();
26          ...
27      }
28  }
```

Listing 4.10: Data element expansion code example (Expanded code)

Data Version Transparency (DVT) theorem

The version transparency implementation for the Data element makes the use of implicit operators and conversion methods, like in the Field element version transparency implementation. However in the Data element, it will not be available a contained object. Instead, when a Data element requires being accessed by a previous version interface, it will invoke the required conversion method and return a new LSP compatible suitable element version.

```
1   public partial class PersonVersion2 : DataElement
2   {
3       public override ElementInfo ElementInfo { get; }
4           = new ElementInfo() { Name = "Person", Version = 2 };
5
6       public PersonIdVersion2 PersonId
7       {
8           get
9           {
10              return Fields["PersonId"].Cast<PersonIdVersion2>();
11          }
12          set
13          {
14              Fields["PersonId"] = value;
15          }
16      }
17
18      public FirstName FirstName
19      {
20          ...
21      }
22
23      public LastName LastName
24      {
25          ...
26      }
27
28      public PersonVersion2()
29      {
30          Fields["PersonId"] = new PersonIdVersion2();
31          ...
32      }
33
34      public static implicit operator Person(PersonVersion2 obj)
35      {
36          var ret = new Person();
37
38          obj.Convert(ret);
39
40          return ret;
41      }
42  }
```

Listing 4.11: Data element expansion code example (Expanded code)

In the listing 4.11 there is presented an example where the `Person` Data element is upgraded by deleting the Field `PersonName` substituted by the new Fields `FirstName` and `LastName` respectively. To perform such conversion LSP compatible a custom method must developed overriding the generic Data element `Convert` method.

```
public partial class PersonVersion3 : DataElement
{
   ...

   public static implicit operator PersonVersion2(PersonVersion3 obj)
   {
      var ret = new PersonVersion2();

      obj.Convert(ret);

      return ret;
   }

   public static implicit operator Person(PersonVersion3 obj)
   {
      return (Person)(PersonVersion2)obj;
   }
}
```

Listing 4.13: Data element multiple conversions (Expanded code)

```
public partial class PersonVersion2 : DataElement
{
   private void Convert(Person obj)
   {
      base.Convert(obj);
      obj.PersonName.Value = FirstName + " " + LastName;
   }
}
```

Listing 4.12: Data element custom `Convert` method (Custom code)

Like in the Field element when an element must convert itself to multiple version it must define a stack of implicit operators, each one to a particular version. Only the first version backward compatibility `Convert` method must be determined and provided, after this first step, the following steps are guaranteed by the previous version conversions. In listing 4.13 is presented an example of this cascading.

Event element implementation

The Event element has the concern of representing a fact or intent, and it should be capable of carrying Data elements to support this fact or intent. In listing 4.14 is presented the base Event element class that supports this concern. Internally it has a `Dictionary` to carry the collection of its body Data elements. Additionally, there is defined a generic `Convert` function that will iterate through the Data collection and the Data collection from another Event element assigning the Data in common with compatible versions. This assigning takes advantage of the underlying Data elements version transparency when accessed through the expanded properties. The properties CorrelationId, Application and Handled with have a supportive nature to the Rule processing engine.

```
public abstract class EventElement : Element
{
   public Guid CorrelationId { get; internal set; } = Guid.NewGuid();

   internal Application Application { get; set; }

   internal bool Handled { get; set; }

   public Dictionary<string, DataElement> ContentData { get; }
      = new Dictionary<string, DataElement>();

   public void Raise()
   {
      Application?.Raise(this);
   }

   protected void Convert(EventElement e)
   {
      e.CorrelationId = CorrelationId;
```

```
20          e.Application = Application;
21          e.Handled = Handled;
22
23          (from orig in ContentData.Values
24           join dest in e.ContentData.Values
25           on orig.ElementInfo.Name equals dest.ElementInfo.Name
26           where orig.ElementInfo.Version >= dest.ElementInfo.Version
27           select orig.ElementInfo.Name).AsParallel().ForAll(
28             result => e.ContentData[result] = ContentData[result]);
29      }
30  }
```

Listing 4.14: Event element code (Base code)

It is evident by the listing 4.15 that the expand process of the Event element is similar to their counterparts Field and Data elements.

```
1  public partial class AskNameCompletedVersion2 : EventElement
2  {
3      public override ElementInfo ElementInfo { get; }
4          = new ElementInfo() { Name = "AskNameCompleted", Version = 2 };
5
6      public PersonVersion2 Person
7      {
8          ...
9      }
10
11     public AskNameCompletedVersion2()
12     {
13         ContentData["Person"] = new PersonVersion2();
14     }
15
16     public static implicit operator AskNameCompleted(AskNameCompletedVersion2 obj)
17     {
18         var ret = new AskNameCompleted();
19         obj.Convert(ret);
20         return ret;
21     }
22 }
```

Listing 4.15: Event element expansion code example (Expanded code)

Data Version Transparency (DVT) theorem

The version transparency implementation of the Event element and its `Convert` method definition follows the same rules as its counterpart Data element. Therefore, it can be infered by the previouslly presented examples for the Data element.

Action element implementation

The Action element has the main concern of performing a single atomic task closely related to an independent change driver. The Action element can consume and produces Data elements to perform its task. However due to the Separation of States (SoS) theorem, the Data elements produced by the Action element are encapsulated into output Event elements (acting as an envelope) that can be sent to the Rule processing engine. Therefore, the base Action element class (Listing 4.16) contains a set for the input data and a set for the output events.

```
1  public abstract class ActionElement : Element
2  {
3      public Dictionary<string, DataElement> InputData { get; }
4          = new Dictionary<string, DataElement>();
5
6      public Dictionary<string, EventElement> OutputEvents { get; }
7              = new Dictionary<string, EventElement>();
8
9      public virtual void Execute() { }
```

```
10  }
```

Listing 4.16: Action element code (Base code)

The Action element expanded code (Listing 4.17) contains for each element of its contained sets (InputData and OutputEvents) a property defined to access these elements in a strongly-typed approach. This expansion technique is very similar to what happen with the other element types when they have contained sets of elements.

```
1  public partial class SayHelloAction : ActionElement
2  {
3     public override ElementInfo ElementInfo { get; }
4        = new ElementInfo() { Name = "SayHelloAction", Version = 1 };
5
6     public Person Person
7     {
8        ...
9     }
10
11    public SayHelloCompleted SayHelloCompleted
12    {
13       ...
14    }
15
16    public SayHelloAction()
17    {
18       InputData["Person"] = new Person();
19       OutputEvents["SayHelloCompleted"] = new SayHelloCompleted();
20    }
21 }
```

Listing 4.17: Action element expansion code example (Expanded code)

However, the functional task that the Action element performs must be defined in a custom code C# partial class (Listing 4.18). Mainly, because the development of this functional task is a responsibility assigned to the Information System (IS) developer. This implementation can be achieved by overriding the Action element `Execute` base method.

```
1  public partial class SayHelloAction
2  {
3     public override void Execute()
4     {
5        Console.WriteLine(
6           string.Format(
7              "V1: Hello {0} and welcome to this new brave world of Normalized Systems! :)",
8              Person.PersonName));
9
10       SayHelloCompleted.Raise();
11    }
12 }
```

Listing 4.18: Action element custom code example (Custom code)

Action Version Transparency (AVT) theorem

The version transparency for the Action element will occur naturally in this architecture. Each Action will be referenced directly in Rules applying a particular version of the Action element. Therefore, a conversion or adaptation of the Action will not occur. The multiple versions of an Action will be grouped together in the IS source code, but their relation will be only structural and not functional. Similarly, with the Action element versioning the Condition element and the Rule element will also have theses kind of versioning properties. Therefore, they also will not have a conversion method to transform these elements between versions. They will be referenced directly.

Condition element implementation

The Condition element is a sub-component of the Rule element, but because it can be considered an independent change driver it was promoted to an element. The main concern of this element is to evaluate the Event elements of a Rule element and vote yes or no to the execution of the Rule elements Actions.

```
1  public abstract class ConditionElement : Element
2  {
3     public Dictionary<string, EventElement> Events { get; }
4              = new Dictionary<string, EventElement>();
5
6     public virtual bool Check() { return true; }
7  }
```

Listing 4.19: Condition element code (Base code)

The Condition element base class (Listing 4.19) contains a set of contained Event elements, that are filled when the Rule is ready to be evaluated. Additionally, it has a `Check` base method that is used to be overriden (Listing 4.20) with a custom implemented method, responsability of the IS developer.

```
1  public partial class CheckPersonName : ConditionElement
2  {
3     public override ElementInfo ElementInfo { get; }
4        = new ElementInfo() { Name = "CheckPersonName", Version = 1 };
5
6     public AskNameCompleted AskNameCompleted
7     {
8        ...
9     }
10
11    public CheckPersonName()
12    {
13       Events["AskNameCompleted"] = new AskNameCompleted();
14    }
15 }
16
17 public partial class CheckPersonName
18 {
19    public override bool Check()
20    {
21       return Regex.IsMatch(AskNameCompleted.Person.PersonName, @"^(\s?[A-Z][^\d]+)+$");
22    }
23 }
```

Listing 4.20: Condition element expanded and custom example code (Expanded & Custom code)

Rule element implementation

The Rule element base class (Listing 4.21) assumes a form of a Event-Condition-Action (ECA) rule. The main purpose of an ECA rule is to listen for incoming Events, when the needed Events are present it must evaluate its Condition, and if its Condition decides yes to the execution of the Rule's Action, then they are executed. The Rule element in this architecture it is very similar to an ECA rule, but instead of one Condition it has multiple Condition elements where they all have to decide yes to the Action execution (And logic). It also has multiple Action elements, instead of only one, that are executed asynchronously in a non-deterministic way.

```
1  public abstract class RuleElement : Element
2  {
3     public Guid CorrelationId { get; internal set; }
4
5     public Application Application { get; internal set; }
6
7     public Dictionary<string, EventElement> Events { get; }
8              = new Dictionary<string, EventElement>();
```

```
9
10      public Dictionary<string, ConditionElement> Conditions { get; }
11              = new Dictionary<string, ConditionElement>();
12
13      public Dictionary<string, ActionElement> Actions { get; }
14              = new Dictionary<string, ActionElement>();
15
16      private bool checkEvents()
17      {
18          ...
19      }
20
21      private bool checkConditions()
22      {
23          ...
24      }
25
26      private void invokeActions()
27      {
28          ...
29      }
30
31      public bool Evaluate()
32      {
33          var ret = false;
34
35          var chkEvents = checkEvents();
36          var chkConditions = (chkEvents ? checkConditions() : false);
37
38          if (chkEvents)
39          {
40              ret = true;
41
42              if (chkConditions)
43              {
44                  invokeActions();
45              }
46          }
47
48          return ret;
49      }
50  }
```

Listing 4.21: Rule element code (Base code)

The way that a Rule check for its incoming Events (Listing 4.22) is by iterating in all of then and check if the `Handled` flag is active. This flag is activated when an Event element is received in the Rule processing engine.

```
1  private bool checkEvents()
2  {
3      return Events.Values.AsParallel().All(e => e.Handled);
4  }
```

Listing 4.22: Check events code (Base code)

To check its Condition elements (Listing 4.23) the Rule element iterates in them, but before executing the `Check` method, it performs a mapping between the Rule's Event elements and the Condition's element Events (Listing 4.23 lines 8-13).

```
1  private bool checkConditions()
2  {
3      if (Conditions.Count() == 0) return true;
4
5      return Conditions.Values.AsParallel().All(
6          c =>
7          {
8              (from ce in c.Events.Values
9               join e in Events.Values
10              on ce.ElementInfo.Name equals e.ElementInfo.Name
11              where e.ElementInfo.Version >= ce.ElementInfo.Version
12              select e.ElementInfo.Name).AsParallel().ForAll(
13                  result => c.Events[result] = Events[result].Clone());
14
15             return c.Check();
16         });
17 }
```

Listing 4.23: Check conditions code (Base code)

To perform the execution of its Action elements, the Rule element iterates in them, but before executing the `Execute` asynchronously, it performs a mapping between the Rule's Event elements content Data elements and the Action input Data elements. It also associates the correlation id and application object to the Action element output Event elements (Listing 4.24).

```
private void invokeActions()
{
   Actions.Values.AsParallel().ForAll(
      a =>
      {
         (from e in Events.Values
          from ed in e.ContentData.Values
          from ad in a.InputData.Values
          where
            ed.ElementInfo.Name == ad.ElementInfo.Name &&
            ed.ElementInfo.Version >= ad.ElementInfo.Version
          select ed).AsParallel().ForAll(
             result => a.InputData[result.ElementInfo.Name] = result.Clone());

         a.OutputEvents.Values.AsParallel().ForAll(
            e =>
            {
               e.CorrelationId = CorrelationId;
               e.Application = Application;
            });

         Task.Run(() => a.Execute());
      });
}
```

Listing 4.24: Execute actions code (Base code)

The Rule element will be similarly expanded like other Elements containing sets of other Elements. It is important to state that this Element will not be expanded with a partial class. Therefore, this Element will not support custom-developed code.

4.2 Application

The Application object is the aggregation of all Elements, notably the Rule elements. In fact, Rule elements are registered in the Application object using the `AddRule<T>` method. These registered Rule elements are mapped in the `rules` and `eventRules` private collections, where the Rule engine will check if a type of Rule element is listening for a particular Event element.

```
public abstract class Application
{
   private readonly Dictionary<ElementInfo, Func<RuleElement>> rules
      = new Dictionary<ElementInfo, Func<RuleElement>>();

   private readonly Dictionary<ElementInfo, HashSet<ElementInfo>> eventRules
      = new Dictionary<ElementInfo, HashSet<ElementInfo>>();

   private readonly Dictionary<Guid, Dictionary<ElementInfo, List<RuleElement>>> waitingRules
      = new Dictionary<Guid, Dictionary<ElementInfo, List<RuleElement>>>();

   private readonly Dictionary<Guid, Dictionary<ElementInfo, Action<EventElement>>> listeners
      = new Dictionary<Guid, Dictionary<ElementInfo, Action<EventElement>>>();

   protected void AddRule<T>()
      where T : RuleElement, new()
   {
      ...
   }

   public void Raise(EventElement e)
   {
      ...
   }

   public async Task<T> Raise<T>(EventElement e, int timeout = Timeout.Infinite)
      where T : EventElement, new()
   {
      ...
```

```
30      }
31
32      public async Task<IEnumerable<T>> Raise<T, TEOF>(EventElement e, int timeout = Timeout.Infinite)
33          where T : EventElement, new()
34          where TEOF : EventElement, new()
35      {
36          ...
37      }
38  }
```

Listing 4.25: Application object code (Base code)

The `Raise` group of methods (Listing 4.25 lines 15-37) will send an Event element to the Rule processing engine without waiting for an response or it could wait for a particular type of raised Event element and even for an array of Event elements until the raise of an `EOF` Event element.

```
1   public class HelloWorld : Application
2   {
3       public HelloWorld()
4       {
5           AddRule<AskNameRule>();
6           ...
7           AddRule<SayHelloCompleteRule>();
8       }
9   }
```

Listing 4.26: Application object expanded code (Expanded code)

The expansion of the Application object is simply the creation of the derived class and the registering of the Rule element types used in the Application to listen for Event elements (Listing 4.26).

Implementing the rule engine

The Rule engine is implemented inside the Application object class. The raise of an Event element triggers the processing of rules. Therefore, the Rule processing engine is embedded in the method that receives the raised Event elements, mainly because it is the most practical solution for this prototype.

```
1   public void Raise(EventElement e)
2   {
3       e.Handled = true;
4
5       var eventinfo = e.ElementInfo;
6
7       if (!waitingRules.ContainsKey(e.CorrelationId))
8           waitingRules[e.CorrelationId] = new Dictionary<ElementInfo, List<RuleElement>>();
9
10      lock (waitingRules[e.CorrelationId])
11      {
```

Listing 4.27: Rule engine source (Base code)

These first lines are concerned with the setup of the Event element processing. If an Event element arrives at this stage with a correlation id that it is not present at the time, the Rule engine must setup the required mechanisms to start a new process, in this case, create a new entry in the `waitingRules` dictionary. To guarantee that this Event element processing is separated from any other Event element of processing, the Rule engine lock this method to only one Event element for each process, isolating it from any other Event element processing at the same process.

```
12          if (eventinfo.Version > 1)
13          {
14              var assembly = e.GetType().Assembly;
15
16              var type = assembly.GetType(
17                  e.GetType().Namespace + "." +
18                  eventinfo.Name +
19                  (eventinfo.Version - 1 > 1 ? "Version" + (eventinfo.Version - 1) : ""));
```

```
20
21              Raise((EventElement)type.Cast(e));
22          }
```

By the DVT theorem if an Event element has a version greater that one there could be Rule elements listening for the Event element lower versions. Therefore, the Event element is cast with the immediate lower version type interface and is re-raised in the system recursively.

```
23          if (listeners.ContainsKey(e.CorrelationId) &&
24              listeners[e.CorrelationId].ContainsKey(eventinfo))
25          {
26              listeners[e.CorrelationId][eventinfo](e);
27          }
```

For each listening Rule type, it will be tested if a waiting rule is suitable to be evaluated. If a suitable Rule element is encountered, then the Rule element is filled with the received Event element and the evaluation process is executed. If a Rule element is considered evaluated then it will be discarded.

```
28          if (eventRules.ContainsKey(eventinfo))
29          {
30              var rulesinfo = eventRules[eventinfo];
31
32              if (!waitingRules.ContainsKey(e.CorrelationId))
33                  waitingRules.Add[]e.CorrelationId] = new Dictionary<ElementInfo, List<RuleElement>>();
34
35              foreach (var ruleinfo in rulesinfo)
36              {
37                  var handled = false;
38
39                  if (waitingRules[e.CorrelationId].ContainsKey(ruleinfo))
40                  {
41                      foreach (var rule in waitingRules[e.CorrelationId][ruleinfo])
42                      {
43                          var v = rule.Events[eventinfo.Name];
44
45                          if (v == null || !v.Handled)
46                          {
47                              rule.Events[eventinfo.Name] = e.Clone();
48                              handled = true;
49
50                              if (rule.Evaluate())
51                                  waitingRules[e.CorrelationId][ruleinfo].Remove(rule);
52
53                              break;
54                          }
55                      }
56                  }
57                  else
58                  {
59                      waitingRules[e.CorrelationId][ruleinfo] = new List<RuleElement>();
60                  }
```

If an Event element was not handled in any waiting Rule element, then it is handled by a newly created Rule element instance. If the Rule element was considered evaluated then that the Rule element is suitable to be discarded. Otherwise, it will be stored in the waitingRules collection.

```
61                  if (!handled)
62                  {
63                      var rule = rules[ruleinfo]();
64
65                      rule.CorrelationId = e.CorrelationId;
66                      rule.Application = this;
67
68                      rule.Events[eventinfo.Name] = e.Clone();
69
70                      if (!rule.Evaluate()) waitingRules[e.CorrelationId][ruleinfo].Add(rule);
71                  }
72              }
73          }
74      }
75  }
```

By using this `Raise` method the proposed architecture can process multiple parallel processes in a

completely asynchronous way, having in all this processing a very low coupling between the elements of the architecture.

4.3 Code expansion

The base library (where the base classes are defined) contains the lower layer of the proposed architecture (Figure 4.1).

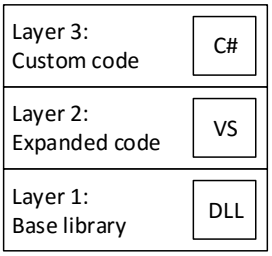


Figure 4.1: 3-Layered software architecture

The middle layer of the architecture is formed by the expanded code; this code can be written by hand by the developer. However, this type of code should be standardized, linear and homogeneous; it is also tedious to write. Therefore, the code expansion will take care of this phase by expanding the defined element in the descriptor file into C# source code. The third layer is the custom code written by the developer to form the required functionalities.

Software elements descriptor

In this architecture implementation, the software elements (Data, Actions, Events, Rules, etc.) are defined in a eXtensible Markup Language (XML) file which is then validated against a XML Schema Definition (XSD) schema definition. This XSD schema defines each element as a `complexType` that contains attributes and sequences of other elements. For example, the Data element (Listing 4.28) extends a base type Element (which provides Name and Version), and contains a sequence of Field Elements. In essence, the XSD schema (Appendix A) enforces the inheritance and aggregations relationships shown earlier in Figure 3.18.

```
<xs:complexType name="DataElement">
   <xs:complexContent>
     <xs:extension base="Element">
       <xs:sequence>
         <xs:element name="Fields">
           <xs:complexType>
             <xs:sequence>
               <xs:element name="Field" type="Element" minOccurs="0" maxOccurs="unbounded"/>
             </xs:sequence>
           </xs:complexType>
         </xs:element>
       </xs:sequence>
     </xs:extension>
   </xs:complexContent>
 </xs:complexType>
```

Listing 4.28: Data element xsd fragment

Implementing template expansion

The process of translating the XML descriptor file into actual C# application code involves a pipeline with a series of stages as illustrated in Figure 4.2:

1. Syntactic checks: the XML descriptor file is saved with a .nsd extension and is validated against the XSD schema.

2. Definition instantiation: the elements in the XML descriptor file are instantiated as C# objects using XML deserialization.

3. Integrity checks: integrity constraints are verified on the object instances with the help of C# and LINQ.

4. Template expansion: the source code is generated with the application of T4 templates to the object instances created from the XML descriptor file.

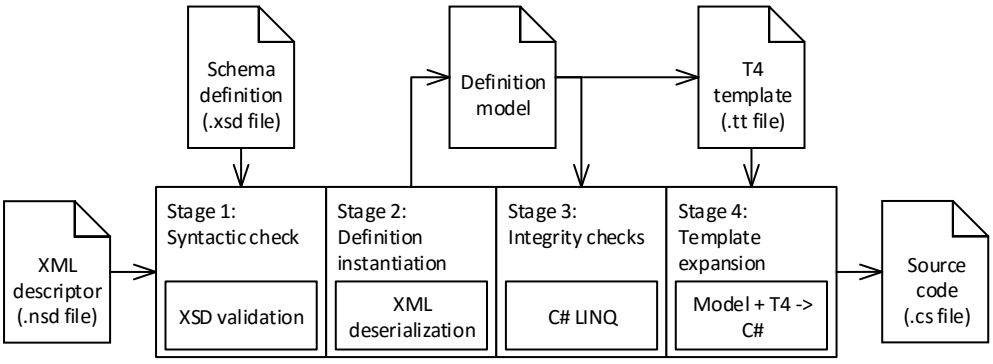


Figure 4.2: Processing stages for code generation

Syntatic checks stage

The first stage of the expansion process is the verification of the NS elements source file. This check is performed using an embedded XSD definition file, verified against the loaded .nsd file. This check only guarantees that the input .nsd file is well formed and can, in fact, be loaded correctly into memory as a model object.

The full XSD definition file is presented in appendix A.

XML deserialization stage

After the first stage, the expansion process can proceed to the next stage. The next stage processes the .nsd input file using a set of class objects (example in listing 4.29) decorated with multiple C# `attributes` to assist in the task of transforming the input XML file into a definition model object. This task is performed by using an .NET `XmlSerializer` object.

```
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("System.Xml", "4.6.1064.2")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
[System.Xml.Serialization.XmlRootAttribute(Namespace = "", IsNullable = true)]
public partial class DataElement : Element
{

   private DeepObservableCollection<Element> fieldsField;

   [Category("Data Element")]
   [System.Xml.Serialization.XmlArrayAttribute(Form = System.Xml.Schema.XmlSchemaForm.Unqualified)]
   [System.Xml.Serialization.XmlArrayItemAttribute("Field", Form =
       System.Xml.Schema.XmlSchemaForm.Unqualified, IsNullable = false)]
   public DeepObservableCollection<Element> Fields
   {
      get
      {
         return this.fieldsField;
      }
      set
      {
         this.fieldsField = value;
         this.fieldsField.CollectionChanged +=
            (sender, e) => { this.RaisePropertyChanged("Fields"); };
         this.RaisePropertyChanged("Fields");
      }
   }
}
```

Listing 4.29: Data element model XML serializable class

Integrity checks stage

A series of integrity constraints are also enforced after XSD validation. For example, one of such constraints is the one indicated in Figure 3.18: if an Event belongs to a Condition, and the Condition belongs to a Rule, then the Event must also belong to the Rule. Some constraints could have been verified using `key` and `keyref` mechanisms available in XSD. However, for more complicated constraints, such as the one above, it becomes more convenient to check them using C# and LINQ. With LINQ (Listing 4.30), it is written queries over object collections in order to verify that the element definitions are free of errors.

```
private static void ruleConditionsCheck(
   IVsGeneratorProgress pGenerateProgress,
   Definitions.Application application)
{
   var query =
      (from rule in application.RuleElements
       from ruleCondition in rule.Conditions
       where
         !application.ConditionElements.Any(
            condition =>
               condition.Name == ruleCondition.Name &&
               condition.Version == ruleCondition.Version)
      select
         new
         {
            Rule = rule,
            Condition = ruleCondition
         });

   foreach (var result in query)
   {
      pGenerateProgress.GeneratorError(
         0, 0,
         string.Format(
            "Condition {0} version {1} on Rule {2} version {3} doesn't " +
            "match with any existent condition on {4} application",
            result.Condition.Name,
            result.Condition.Version,
            result.Rule.Name,
            result.Rule.Version,
            application.Name), 0xFFFFFFFF, 0xFFFFFFFF);
   }
}
```

Listing 4.30: Integrity check example(A Rule must reference an existing Condition)

```
<#@ template language="C#" #>
<#@ assembly name="System.Core" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="System.Text" #>
<#@ import namespace="System.Collections.Generic" #>
   public partial class <#= model.FullName #> : NormalizedSystems.Net.ActionElement
   {
<#  foreach(var data in model.InputData) { #>
      public <#= data.FullName #> <#= data.Name #> { get; set; }
<#  } #>
<#  foreach(var evt in model.OutputEvents) { #>
      public <#= evt.FullName #> <#= evt.Name #> { get; set; }
<#  } #>
}
```

Listing 4.31: Action element T4 template

If it encountered any type of integrity violation, the expander will interrupt its code generation process and report the error back to the Visual Studio IDE.

Template expansion stage

Once the first three stages are complete, the element definitions have been transformed into C# objects in memory. We then use a set of T4 templates (one for each type of element) to generate the source code for the application. Initially, it was considered the option of using eXtensible Stylesheet Language for Transformation (XSLT) to generate the source code from the XML descriptor file, but T4 revealed to be more adequate in this code-generation scenario.

In essence, a T4 template is a mixture of text blocks and control blocks that specify how to generate an output text file. For example, a T4 template can generate multiple repetitions of the same text block by means of a control block with a loop. The generated output can be a text file of any kind, such as a Web page, a resource file, or program source code in any language. A T4 template can therefore be seen as a "program" that generates C# source code.

In the architecture T4 templates, both the text blocks (the source code to be generated) and the control blocks (the instructions to generate the code) are written in C#. The T4 template is processed by special component available in Visual Studio, called `TextTemplatingFilePreprocessor`, a custom tool like the developed in this work. Given the T4 template and a set of C# objects (the element definitions used as a model to the template), the `TextTemplatingFilePreprocessor` generates the source code for the application based on the properties of those objects.

Finally, the generated C# source code file will be attached as a child file of the input .nsd file. Inside will have all the code artifacts needed to structure a NS application in this work architecture.

Customizations and harvesting

The generated code is a skeleton for the target application, and the developer will need to add custom code to the project. Rather than changing the generated code (which will be re-generated at each new change to the XML descriptor file), any custom code is kept in separate source files. The C# language provides a convenient mechanism to allow this code separation: using *partial classes* [39] it is possible to have a class definition split across multiple source files.

All of the generated C# classes are declared as partial to provide the possibility of being extended with additional source files that the developer adds to the project. This method of using partial classes when dealing with generated code is available since .NET Framework 2.0 and is a well-established practice nowadays. For example, it is used by WinForms, WPF, ASP.NET Web Forms, and many other tools and designers available in the Visual Studio environment.

Using C# with partial classes avoids having to deal with the problem of *harvesting* [25], which consists in extracting any custom code from a previous version and re-inserting it into a new version of the application. In the developed framework, harvesting is unnecessary for the following reasons:

- there is no need to edit the generated source code to inject custom code;
- there is no need to have anchor points because the custom code resides in a separate file;
- the custom code will not be lost with a new expansion of the descriptor file.

In other languages, which do not have the ability to split the definition of a class into multiple source files, it will be necessary to implement anchor points and harvesting.

4.4 Visual Studio extension

The tools created for this work are implemented as an extension to Visual Studio .NET 2015. It is constituted by three components. The custom tool: that is responsible for expanding the descriptor into C# source code. The visual designer responsible for providing a visual editor to the developer for editing descriptor files. Also, lastly, the item template responsible for adding a new NS descriptor file, completely configured, into the project.

Visual designer

Although the software elements are to be defined in an XML descriptor file, it is certain that is not intended to require the developer to write such XML file by hand. Therefore, it was developed a Visual Studio extension that provides a custom designer to define those software elements with a graphical user interface. This designer is shown in Figure 4.3.

Basically, this custom designer is a Windows Presentation Foundation (WPF) user control that uses a view model that is closely related to the Visual Studio project subsystem. This view model will have an instance of the definitions model object connected to the edited .nsd file. With this arrangement, it is possible to edit the descriptor file both in the custom designer and directly in XML form, while keeping both views synchronized with each other. It is also possible to edit the file externally while maintaining synchronization between all editors.

Item template

Along with this extension, it was also created a custom item template for the NS Application. This template appear in the “Add New Item” dialog box in Visual Studio (Figure 4.4), under a new “Normalized

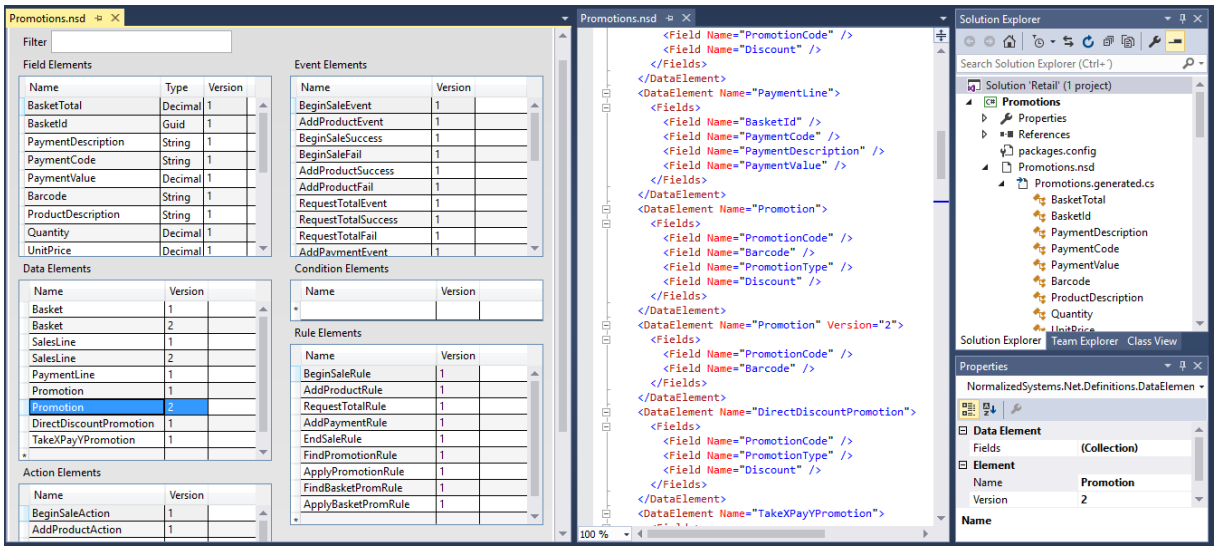


Figure 4.3: Custom designer for creating an XML descriptor file

Systems" category that is present in the C# project/item template list.

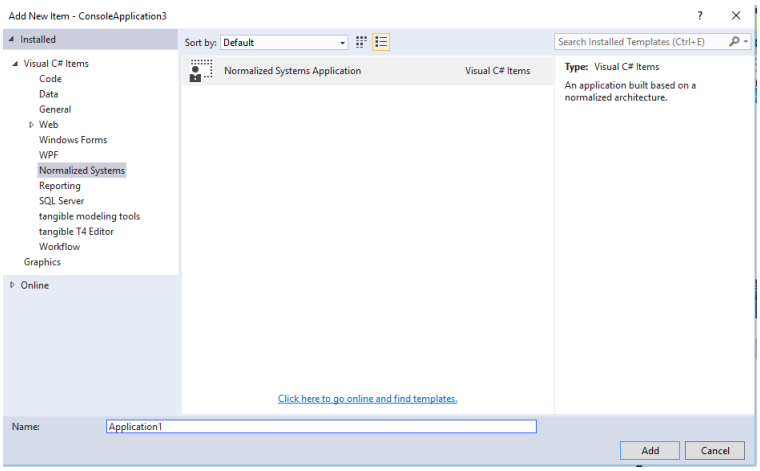


Figure 4.4: Visual Studio "Add new item" dialog selecting a new NS Application item.

Custom tool

The four stages depicted in Figure 4.2 are implemented inside a custom tool that was developed for Visual Studio. This custom tool is named "NormalizedSystemExpander", and it is invoked whenever there is a change to the XML descriptor file (Figure 4.5). In this case, it re-generates the application source code automatically.

The custom tool was developed as a Component Object Model (COM) component that implements the `IVsSingleFileGenerator` interface. Through this interface, a custom tool transforms a single input file into a single output file. When the input file is saved, the custom tool is instantiated and Visual Studio calls its `Generate` method, passing a reference to an `IVsGeneratorProgress` callback interface that the tool can use to report its progress. This is used to provide feedback on the expansion process.

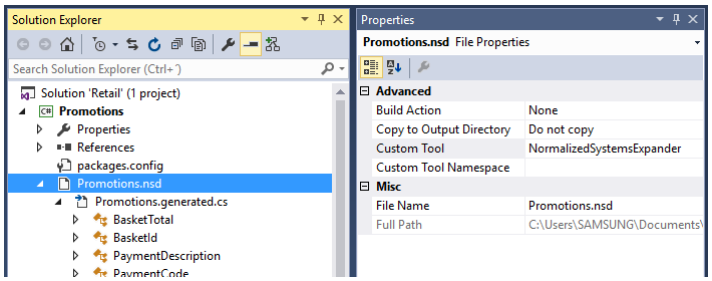


Figure 4.5: Visual Studio "NormalizedSystemExpander" custom tool created for this NS architecture.

The output generated by the custom tool (i.e. the application source code) is added to the project together with a dependency to the input descriptor file.

4.5 Summary

Originally, the normalized systems development framework was implemented over a J2EE platform based on Java technology and structured as a 4-Tier Web application architecture [5]. However, the architecture implementation presented in this chapter is suitable to develop multiple types of application, e.g., CLI, Desktop, Web applications, Web Services and others.

The only platform non-functional requirement for the implemented architecture is the .NET framework. Additionally, the C# programming language is also a non-functional requirement for the development of custom code on behalf of the developer.

In this section, it is described each element implementation in all three layers of the architecture: base library; expanded code; custom code (Section 4.3). This description of each element is based on a simple example application created for the purpose of demonstrating the architecture. This example "Hello world" application has the simple task of asking the user name's and print a message with the name contained in it.

In this architecture implementation, the software elements (Data, Actions, Events, Rules, etc.) are defined in a XML file which is then validated against a XSD schema definition. This XSD schema defines each element as a `complexType` that contains attributes and sequences of other elements.

The process of translating the XML descriptor file into actual C# application code involves a pipeline with a series of stages as illustrated in Figure 4.2:

1. Syntactic checks: the XML descriptor file is saved with a .nsd extension and is validated against the XSD schema.

2. Definition instantiation: the elements in the XML descriptor file are instantiated as C# objects using XML deserialization.

3. Integrity checks: integrity constraints are verified on the object instances with the help of C# and LINQ.

4. Template expansion: the source code is generated with the application of T4 templates to the object instances created from the XML descriptor file.

The tools created for this work are implemented as an extension to Visual Studio .NET 2015 and are constituted by three components. The custom tool: that is responsible for expanding the descriptor into C# source code. The visual designer responsible for providing a visual editor to the developer for editing descriptor files. Also, lastly, the item template responsible for adding a new NS descriptor file, completely configured, into the project.

Chapter 5

Case study

To illustrate how the proposed architecture can be applied in practice, it was applied an example inspired by previous experience in the retail industry. For many years, the retail industry was a relatively straightforward type of business, but the introduction of promotional campaigns and customer loyalty programs significantly increased the complexity of this business. Nowadays, the retail industry is characterized as being a highly dynamic business environment, where there is a constant struggle to maintain a leading edge over competitors. In terms of IT and supporting systems, it is common to spend several weeks or even months on the implementation of a new feature, to be used in a customer engagement campaign that lasts only two weeks or even less. NS theory seems to be a perfect fit to deal with the fast pace at which requirements change in this scenario, especially when focusing on the promotions and customer loyalty systems.

5.1 Sales process without promotions

In the first stage of this case study, it was directed to a sales process without any promotional campaigns. The goal is to identify the core software elements that will also be used in subsequent stages, where the sales process grows in complexity as promotional campaigns are added.

The process will have to interact with an external point-of-sale (POS) system, which raises different kinds of events:

- `BeginSaleEvent` – this marks the beginning of a sales transaction, and is equivalent to the beginning of a new transaction in a database system.
- `AddProductEvent` – this marks the moment when a product is added to the sales basket. It can also be used to remove products from the basket, if the quantity is negative. Usually, a product is introduced (or cancelled) by scanning a barcode or by manual input on the POS keypad. Other variants involve the manual introduction of quantity, dynamic barcodes (for multiple units of the same product in a single pack) and weight scales (for products sold by weight).
- `RequestTotalEvent` – when all products have been introduced in the basket and the customer

is ready to pay, the POS system requires an intermediate operation with the purpose of calculating all the sales subtotals and present the final value to the customer.

- `AddPaymentEvent` – when the sale has been paid, the POS system requires the execution of an operation to register the payment and update the basket accordingly.
- `EndSaleEvent` – this marks the end of a sales transaction, and is equivalent to a commit in a database system.

Table 5.1 shows these Events together with the Rules that they fire, the Actions invoked by those Rules, and the output Events raised by those Actions.

POS Event	Rule	Action	Raised Events
BeginSaleEvent	BeginSaleRule	BeginSaleAction	BeginSaleSuccess BeginSaleFail
AddProductEvent	AddProductRule	AddProductAction	AddProductSuccess AddProductFail
RequestTotalEvent	RequestTotalRule	RequestTotalAction	RequestTotalSuccess RequestTotalFail
AddPaymentEvent	AddPaymentRule	AddPaymentAction	AddPaymentSuccess AddPaymentFail
EndSaleEvent	EndSaleRule	EndSaleAction	EndSaleSuccess EndSaleFail

Table 5.1: Elements for the sales process

At this stage, the process is mostly concerned with the management of data in the basket. These data can be represented as one or more Data elements as shown in Figure 5.1. The AddProductAction appends a new SalesLine to the Basket, and the AddPaymentAction sets the PaymentLine in the Basket.

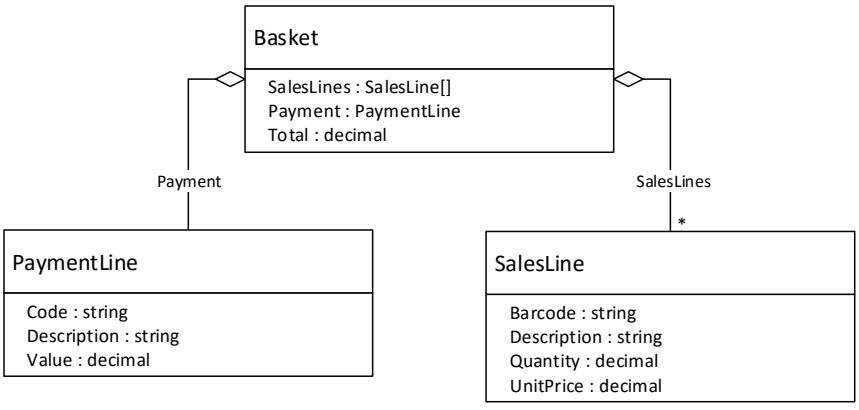


Figure 5.1: Data elements for the sales basket (simplified)

5.2 Sales process with direct discounts

In the second stage of this case study, the sales process was upgraded to include the possibility of having direct discounts on some products. These are the simplest kind of promotions in the retail sector, so they are a good candidate to illustrate an incremental evolution of the previous process.

Typically, the way to implement direct discounts in the retail industry is to have a list of existing promotions and, given a particular basket, to look for applicable promotions to the products contained in that basket. For this purpose, a new Data element is added to represent a direct promotion. Additionally, the Basket will now have two more attributes, and each SalesLine may have a reference to a Promotion. The new data elements are illustrated in Figure 5.2.

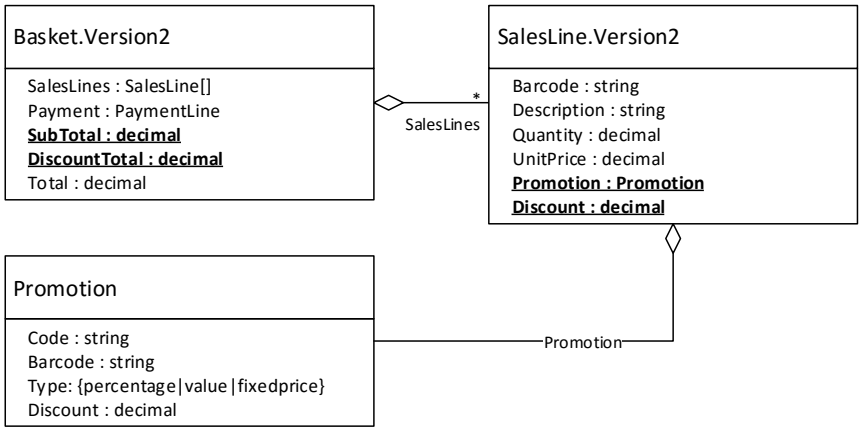


Figure 5.2: Data elements for direct discount promotions

The implementation of direct discounts requires the introduction of two new Rules. A first Rule checks whether there is a promotion for a product (SalesLine) that has just been added to the Basket. If a promotion is found, then a second Rule applies the discount to that SalesLine. Table 5.2 shows the new elements to be added to the sales process.

Event	Rule	Action	Raised Events
AddProductSuccess	FindPromotionRule	FindPromotionAction	FindPromotionSuccess FindPromotionFail
FindPromotionSuccess	ApplyPromotionRule	ApplyPromotionAction	ApplyPromotionSuccess ApplyPromotionFail

Table 5.2: Additional elements for direct discounts

5.3 Sales process with "take x, pay y" promotions

In the third stage of this case study, it is introduced the possibility of having a different type of promotion which takes into account multiple sales lines in the basket. This kind of promotions is usually designated basket promotions.

The "take x, pay y" promotion is a well-known basket promotion that allows a customer to buy x units of a certain product but pay only a fraction (y/x) of their total price. This is implemented by having two dedicated fields: one is the TakeQuantity field which stores the minimum product quantity that must exist in the basket in order to apply this promotion; the other field is OfferQuantity, which defines how much quantity is offered to the customer every time the promotion is applied. Applying this promotion results in a discount being added to one or more sales lines.

Since individual sales lines can be canceled during a sale, the application of this promotion must occur at a later stage of the sale transaction, when the amount of products in the basket can no longer change. Therefore, this promotion must be applied on the RequestTotalEvent, after all products have been scanned and the basket is considered closed.

With the introduction of this new promotion, the possibility of having multiple promotions targeted at the same product arises and a conflict can occur. Therefore, promotions must be prioritized, which requires the use of a Priority field. The calculation of direct discount promotions as products are being scanned can be maintained but, when the total is requested, other promotions can prevail, overriding the existing direct discount promotions.

For example, given two promotions targeted at the same product – a direct promotion offering a 10% discount with priority 2; and a "take 3, pay 2" promotion with priority 1 (higher priority) – then the direct discount of 10% will be offered at each input of such product, but if there are 3 units or more of that product in the basket, the second promotion will replace the first promotion when the total is requested.

The implementation of basket promotions requires a refactoring of the data elements in order to introduce a new version of the Promotion data element that can support both types of promotions, as illustrated in Figure 5.3.

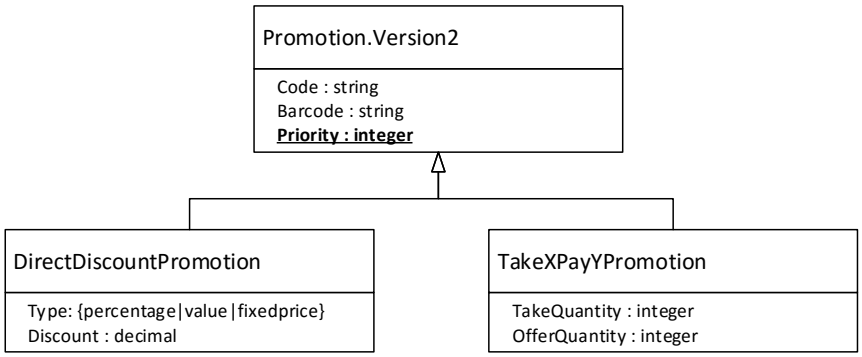


Figure 5.3: Data elements for "take x, pay y" promotions

In particular, the new Promotion element contains the fields that are common to both promotions (Code for the promotion code, and Barcode for the product code) as well as an additional field to indicate Priority. For direct promotions, the Type and Discount fields continue to exist, but are now placed in a subclass of Promotion. The same happens with "take x, pay y" promotions, which have their own separate subclass with TakeQuantity and OfferQuantity fields.

Additionally, both FindPromotionAction and ApplyPromotionAction should be upgraded with a new version to support the changes made to the Promotion Data element and the inclusion of the DirectDiscountPromotion Data element. Both versions will function in tandem.

Finally, to support basket promotions it is necessary to intercept the RequestTotalEvent and execute intermediate actions to find and apply matching promotions based on product quantities, before the actual total is calculated by RequestTotalAction. Table 5.3 shows the changes to the process.

Event	Rule	Action	Raised Events
RequestTotalEvent	FindBasketPromRule	FindBasketPromAction	FindBasketPromSuccess FindBasketPromFail
FindBasketPromSuccess	ApplyBasketPromRule	ApplyBasketPromAction	ApplyBasketPromSuccess ApplyBasketPromFail
ApplyBasketPromSuccess	RequestTotalRule	RequestTotalAction	RequestTotalSuccess RequestTotalFail

Table 5.3: Changes to support basket promotions

5.4 Loyalty promotions

In the fourth stage of this case study, it is introduced the possibility of having targeted promotions to specific customers. This kind of promotions is usually designated loyalty promotions. In loyalty systems, the customer is identified during the sale process, and this identification registers sale data associated with the customer.

This data can be used later to offer exclusive benefits to the customer. The objective its to better know the recurring customers and improve the possibility of previously visiting customers to continue to spend money in the organization.

The introduction of a new “Benefit” entity is useful to represent offers that a customer can take advantage while visiting the organization stores. Benefits can be associated with customers in a many-to-one relationship while promotions can be linked to a benefit. In this case study, the offer represented by benefits will be a promotion discount. Therefore, promotions with a benefit will trigger only if a customer has that benefit associated.

In figure 5.4 it is presented the data elements involved in this fourth stage implementation.

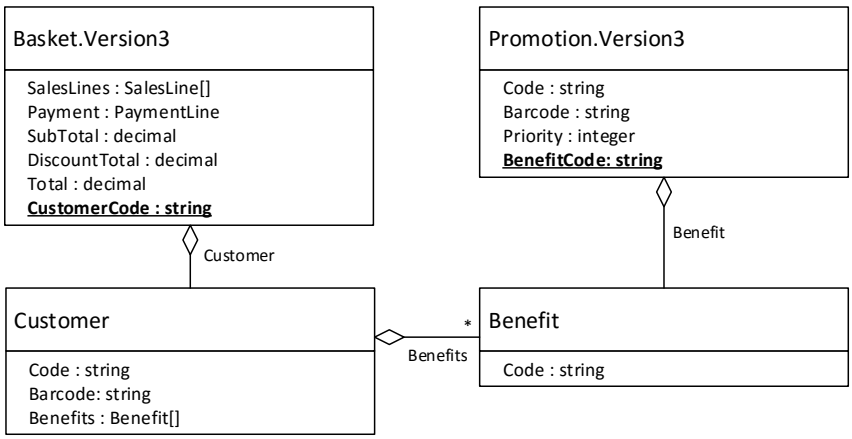


Figure 5.4: Data elements for loyalty promotions

At this stage, it is clear that the number of changes is constrained by a very limited set of changes. There is only the addition of one Field in the Basket (to support the inclusion of a Customer) and Promotion (to support the inclusion of a Benefit) Data elements, and also the inclusion of the Customer and Benefit Data elements.

Additionally like in the third stage, there is the necessity to upgrade some Action elements to support the functional changes required for this stage.

Both, FindBasketPromAction and FindPromotionAction Action elements should be updated to support the inclusion of the Customer during the sale and the Benefit code definition in the Promotion Data element. Additionally to the prior functional requirements, these Action elements should make a selection of the promotions taking into account the relationship between the customer present in the sale versus the related benefits and the promotions contained in the set of benefits that the customer has.

Finally, to support loyalty promotions it is necessary to set the customer during the sale process. To achieve this objective, the SetCustomerEvent is sent by the Point-Of-Sale (POS), which is handled by the application by the SetCustomerRule, which invokes the SetCustomer Action to set the Customer Data element in the Basket Data element. Table 5.4 shows the changes to the process.

Event	**Rule**	**Action**	**Raised Events**
SetCustomerEvent	SetCustomerRule	SetCustomerAction	SetCustomerSucess SetCustomerFail

Table 5.4: Changes to support loyalty promotions

5.5 Summary

To illustrate how the proposed architecture can be applied in practice, it was applied an example inspired by previous experience in the retail industry. This case study is divided in four different stages:

- *Sales process without promotions*: It is focused on a sales process without any promotional campaigns. The goal is to identify the core software elements that will also be used in subsequent stages, where the sales process grows in complexity as promotional campaigns are added.
- *Sales process with direct discounts*: The sales process is upgraded to include the possibility of having direct discounts on some products. These are the simplest kind of promotions in the retail sector, so they are a good candidate to illustrate an incremental evolution of the previous process.
- *Sales process with "take x, pay y" promotions*: It is introduced the possibility of having a different type of promotion which takes into account multiple sales lines in the basket. The "take x, pay y" promotion is a well-known basket promotion that allows a customer to buy x units of a certain product but pay only a fraction (y/x) of their total price.
- *Loyalty promotions*: It is introduced the possibility of having targeted promotions to specific customers. In loyalty systems, the customer is identified during the sale process, and this identification registers sale data associated with the customer. This data can be used later to offer exclusive benefits to the customer.

The conclusion to get from this case study is that by using this architecture, it was possible to comply with all the desired properties expected from a normalized system. The most remarkable property

experienced in this chapter was the evolvability capacity that this architecture offered to this case study application. At each stage, the changes needed to support the new requirements was systematically limited to only the strictly necessary modifications, leaving the rest of the system untouched. Revealing an excellent stability when facing volatile requirement changes.

Chapter 6

Conclusion

This work proposed a new architecture for Normalized Systems inspired by the idea of using ECA rules to implement process behavior and achieve a fine-grained modular structure. The original Normalized Systems (NS) architecture uses a state machine approach to implement process logic. There has been an ongoing debate in the past two decades about the advantages of implementing business processes with rule-based vs. graph-based approaches. Without getting too much into that debate, it was observed that a rule-based approach fits better with the fundamental principles of NS theory, which strives for low-coupling and aims at a fine-grained structure to support changes.

Using an application scenario that draws on practical experience in the retail industry, we have illustrated how this rule-based approach facilitates the accommodation of changes to a business process and supports its increasing complexity over time. Our current and future work is focused on a prototype implementation of the proposed architecture by taking advantage of specific features of the C# language and the extensibility of its development environment in order to bring the benefits of NS theory to a wider range of applications.

6.1 Contributions

The main contribution of this work is the development of a framework that enforces the concepts and principles of NS theory by using them as its foundation. This framework exhibits the same theoretic stability as intended by the NS original architecture.

One of the issues mentioned with the original NS architecture is that its reference implementation is not publicly available (not even in a closed source alternative). Therefore, the result of this dissertation work will be publicly available at: `https://github.com/pedrosimoes79/normalizedsystems.net`.

Another contribution is the use of rule and events instead of the simpler approach of a state machine made by the original authors. The use of rules and events can be regarded as harder to control and understand, but the gained flexibility and decoupling counterbalance the mentioned drawbacks. It was apparent that this approach can maintain the system stability and evolvability and at the same time comply with all the original counterpart theoretical basis.

Regarding the implementation of this work, a complete base framework library was developed in the form of a Dynamic-Link Library (DLL) library or a `NuGet` package that can be published in the official `NuGet` package repository or used as a library in a standard C# project. In fact, it can be possible to write a normalized application only by reference this library.

Although the developed base library is capable of producing a complete normalized application, this process can be tedious and error prone. In this dissertation work, we also implemented tools to cope with this help with this issue. These tools include a code-generation expander tool that works in an integrated way with the Visual Studio IDE to expand a descriptor file into standard pre-defined C# source-code (leveraged by Text Template Transformation Toolkit (T4) templates). We also implemented a visual aid tool to edit the descriptor files and a Visual Studio item template.

In the end, the developer is provided with a set of tools that allows the development of a normalized application by using a high abstraction level tool, which is agnostic to the underlying technology, and produces a complete skeleton of the application architecture without the need to write any code.

6.2 Future work

There is great potential for future work in this topic. This document initiated the developement (at a conceptual and practical level) of a new NS framework and confirmed that it is possible to create such framework only by following the NS theory and according to the theory concepts and principles.

An interesting task to extend this proposed framework is the implementation of the architecture in other programming languages, like C/C++ or Java, demonstrating that the architecture is platform-agnostic. This extension can be easily developed by modifying the expander in order to expand source code in other programming languages.

The tools created in this work have a strong relationship with the Visual Studio .NET 2015 and the base .NET Framework. Another interesting development is the separation of the expander and visual designer from Visual Studio .NET, in order to be possible to integrate it with other IDE's or tools, e.g., Linux and Mono. This development can be accomplished by performing only some simple refactoring in these code of these tools.

An interesting avenue for future work is to develop more applications in real-life Information System (IS) projects. At present, the framework is only a prototype, and to be used in production projects it will be required to improve some aspects, namely its resilience, performance, and reliability.

Bibliography

[1] R. Endl, G. Knolmayer, and M. Pfahrer, “Modeling processes and workflows by business rules,” *SWORDIES Report*, vol. Nr. 18, pp. 1–10, 1998.

[2] M. McIlroy, “Mass produced software components,” in *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968*, 1968, pp. 138–155.

[3] D. Parnas, “Software aging,” in *16th International Conference on Software Engineering*, 1994, pp. 279–287.

[4] M. M. Lehman, “Programs, life cycles, and laws of software evolution,” *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980.

[5] H. Mannaert and J. Verelst, “Normalized systems: Re-creating information technology based on laws for software evolvability,” *Koppa, Belgium*, 2009.

[6] G. Kappel, B. Proll, S. Rausch-Schott, and W. Retschitzegger, “TriGSflow: Active object-oriented workflow management,” in *Proceedings of the 28th Hawaii International Conference on System Sciences*, vol. 2, 1995, pp. 727–736.

[7] S. Ceri, P. Grefen, and G. Sánchez, “WIDE – a distributed architecture for workflow management,” in *7th International Workshop on Research Issues in Data Engineering*, 1997.

[8] A. Geppert and D. Tombros, “Event-based distributed workflow execution with eve,” *SWORDIES Report*, vol. Nr. 17, pp. 1–16, 1998.

[9] A. Goh, Y.-K. Koh, and D. Domazet, “ECA rule-based support for workflows,” *Artificial Intelligence in Engineering*, vol. 15, no. 1, pp. 37–46, 2001.

[10] J. Bae, H. Bae, S.-H. Kang, and Y. Kim, “Automatic control of workflow processes using ECA rules,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 8, pp. 1010–1023, 2004.

[11] F. Bry, M. Eckert, P.-L. Pătrânjan, and I. Romanenko, “Realizing business processes with ECA rules: Benefits, challenges, limits,” in *Principles and Practice of Semantic Web Reasoning*, ser. LNCS, vol. 4187. Springer, 2006, pp. 48–62.

[12] K. Ven, D. V. Nuffel, and P. Huysmans, “Experiences with the automatic discovery of violations to the normalized systems design theorems,” *International Journal on Advances in Software*, vol. 4, no. 1–2, pp. 46–60, 2011.

[13] P. D. Bruyn, P. Huysmans, G. Oorts, D. V. Nuffel, H. Mannaert, and J. Verelst, “Incorporating design knowledge into the software development process using normalized systems theory,” *International Journal on Advances in Software*, vol. 6, no. 1, pp. 181–195, 2013.

[14] P. Huysmans, D. Bellens, D. Van Nuffel, and K. Ven, “Designing enterprise architectures based on systems theoretic stability,” in *ICE-B 2010 International Conference on e-Business*, 2010, pp. 157–162.

[15] P. Huysmans, J. Verelst, H. Mannaert, and A. Oost, “Integrating information systems using normalized systems theory: Four case studies,” in *IEEE 17th Conference on Business Informatics (CBI)*, vol. 1, July 2015, pp. 173–180.

[16] H. Mannaert, P. D. Bruyn, and J. Verelst, “Exploring entropy in software systems: Towards a precise definition and design rules,” *The Seventh International Conference on Systems (ICONS 2012)*, no. c, pp. 93–99, 2012.

[17] P. De Bruyn, G. Dierckx, and H. Mannaert, “Aligning the normalized systems theorems with existing heuristic software engineering knowledge,” in *ICSEA 2012, The Seventh International Conference on Software Engineering Advances*, November 2012, pp. 84–89.

[18] G. Oorts, K. Ahmadpour, H. Mannaert, and J. Verelst, “Easily evolving software using normalized system theory a case study,” in *ICSEA 2014, The Ninth International Conference on Software Engineering Advances*, 2014, pp. 322–327.

[19] H. Mannaert, J. Verelst, and K. Ven, “Towards evolvable software architectures based on systems theoretic stability,” *Software: Practice and Experience*, vol. 42, no. 1, pp. 89–116, 2012.

[20] D. van der Linden, H. Mannaert, W. Kastner, V. Vanderputten, H. Peremans, and J. Verelst, “An OPC UA interface for an evolvable ISA88 control module,” in *Emerging Technologies Factory Automation (ETFA), 2011 IEEE 16th Conference on*, September 2011, pp. 1–9.

[21] P. De Bruyn, H. Mannaert, and J. Verelst, “Towards organizational modules and patterns based on normalized systems theory,” in *The Ninth Internation Conferece on Systems (ICONS 2014)*, 2014, pp. 106–115.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*, 1st ed. Addison-Wesley Professional, 1994.

[23] P. De Bruyn, D. Van Nuffel, J. Verelst, and H. Mannaert, “Towards applying normalized systems theory implications to enterprise process reference models,” in *Advances in Enterprise Engineering VI*, ser. Lecture Notes in Business Information Processing. Springer, 2012, vol. 110, pp. 31–45.

[24] J. Wetherbee, R. Kodali, C. Rathod, and P. Zadrozny, *Beginning EJB 3: Java EE 7 Edition*, ser. Expert's voice in Java. Apress, 2013.

[25] G. Oorts, P. Huysmans, P. De Bruyn, H. Mannaert, J. Verelst, and A. Oost, "Building evolvable software using normalized systems theory: A case study," in *47th Hawaii International Conference on System Sciences*, 2014, pp. 4760–4769.

[26] R. Lu, "A survey of comparative business process modeling approaches," *Business Information Systems*, vol. 4439, p. 82–94, 2007.

[27] T. DeMarco, *Structured Analysis and System Specification*. Prentice Hall, 1979.

[28] C. Gane and T. Sarson, *Structured Systems Analysis: Tools and Techniques*. Prentice Hall, 1979.

[29] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-oriented modeling and design*. Prentice Hall, 1991.

[30] D. Georgakopoulos, M. Hornick, and A. Sheth, "An overview of workflow management: From process modeling to workflow automation infrastructure," *Distributed and Parallel Databases*, vol. 3, no. 2, pp. 119–153, April 1995.

[31] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*. Addison Wesley, 1999.

[32] D. Miers and S. A. White, *BPMN Modeling and Reference Guide*. Future Strategies Inc., 2008.

[33] M. Fowler, *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[34] W. M. P. van der Aalst, "The application of Petri nets to workflow management," *Journal of Circuits, Systems and Computers*, vol. 8, no. 1, pp. 21–66, 1998.

[35] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros, "Workflow patterns," *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 5–51, 2003.

[36] W. M. P. van der Aalst, A. J. Mooij, C. Stahl, and K. Wolf, "Service interaction: Patterns, formalization, and analysis," in *Formal Methods for Web Services*, ser. LNCS, vol. 5569. Springer, 2009, pp. 42–88.

[37] B. H. Liskov and J. M. Wing, "A behavioral notion of subtyping," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 6, pp. 1811–1841, Nov. 1994.

[38] H. Mannaert, J. Verelst, and K. Ven, "The transformation of requirements into software primitives: Studying evolvability based on systems theoretic stability," *Science of Computer Programming*, vol. 76, no. 12, pp. 1210–1222, 2011.

[39] K. Czarnecki, M. Antkiewicz, and C. H. P. Kim, "Multi-level customization in application engineering," *Communnications of the ACM*, vol. 49, no. 12, pp. 60–65, December 2006.

Appendices

Appendix A

XSD descriptor file definition

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:simpleType name="Identifier">
    <xs:restriction base="xs:string">
      <xs:pattern value="^[A-Z][a-zA-Z0-9]*$" />
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="FieldType">
    <xs:restriction base="xs:string">
      <xs:pattern value="Byte|Int8|UInt8|Short|UShort|Int16|UInt16|Integer|
                         Unsigned|Int32|UInt32|Long|ULong|Int64|UInt64|Float|
                         Double|Decimal|Date|Time|DateTime|Timespan|Guid|
                         String|ByteArray" />
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="Element">
    <xs:attribute name="Name" type="Identifier" use="required"/>
    <xs:attribute name="Version" type="xs:unsignedInt" default="1"/>
  </xs:complexType>

  <xs:complexType name="FieldElement">
    <xs:complexContent>
      <xs:extension base="Element">
        <xs:attribute name="Type" type="FieldType"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="DataElement">
    <xs:complexContent>
      <xs:extension base="Element">
        <xs:sequence>
          <xs:element name="Fields">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="Field" type="Element" minOccurs="0" maxOccurs="unbounded"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="EventElement">
    <xs:complexContent>
      <xs:extension base="Element">
        <xs:sequence>
          <xs:element name="ContentData">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="Data" type="Element" minOccurs="0" maxOccurs="unbounded"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
```

```
<xs:complexType name="ActionElement">
  <xs:complexContent>
    <xs:extension base="Element">
      <xs:sequence>
        <xs:element name="InputData">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Data" type="Element" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="OutputEvents">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Event" type="Element" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="ConditionElement">
  <xs:complexContent>
    <xs:extension base="Element">
      <xs:sequence>
        <xs:element name="Events">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Event" type="Element" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="RuleElement">
  <xs:complexContent>
    <xs:extension base="Element">
      <xs:sequence>
        <xs:element name="Events">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Event" type="Element" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="Conditions">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Condition" type="Element" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="Actions">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Action" type="Element" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:element name="Application">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="FieldElements">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="FieldElement" type="FieldElement" minOccurs="0" maxOccurs="unbounded"
                />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="DataElements">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="DataElement" type="DataElement" minOccurs="0" maxOccurs="unbounded" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="EventElements">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="EventElement" type="EventElement" minOccurs="0" maxOccurs="unbounded"
                />
```

```
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="ActionElements">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="ActionElement" type="ActionElement" minOccurs="0"
              maxOccurs="unbounded" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="ConditionElements">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="ConditionElement" type="ConditionElement" minOccurs="0"
              maxOccurs="unbounded" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="RuleElements">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="RuleElement" type="RuleElement" minOccurs="0" maxOccurs="unbounded" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
  <xs:attribute name="Name" type="Identifier" use="required"/>
 </xs:complexType>
</xs:element>
</xs:schema>
```

Appendix B

Field data types definition

Field Data Type	C# Data Type
Byte	uint
UInt8	uint
Int8	sbyte
Int16	short
Short	short
UInt16	ushort
UShort	ushort
Int32	int
Integer	int
UInt32	uint
Unsigned	uint
Int64	long
Long	long
UInt64	ulong
ULong	ulong
Float	float
Double	double
Decimal	decimal
DateTime	DateTime
Date	DateTime
TimeSpan	TimeSpan
Time	TimeSpan
Guid	Guid
String	string
ByteArray	byte[]

Table B.1: Platform independent field data types