A Development Framework for Normalized Systems (Extended abstract)

Pedro Filipe Pires Simões
Instituto Superior Técnico (IST)
University of Lisbon, Portugal
E-mail: pedro.simoes@tecnico.ulisboa.pt

***Abstract*—The theory of Normalized Systems 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. 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 and can be used to implement process behavior based on state machines. In this work, we propose a different approach to the implementation of process behavior in Normalized Systems. Specifically, the approach is based on the use of rules and events as fine-grained elements to build complex behavior. Our previous experience in the retail industry suggests that such approach is capable of dealing with the increasing complexity of business processes in that sector. We also discuss some details regarding the implementation of this new architecture on top of a popular technological platform and its supporting tools.**

***Keywords*—Normalized Systems, Workflow and Business Processes, ECA Rules**

I. Introduction

Normalized Systems (NS) [1] is a software engineering approach to the development of business information systems, with a special focus on supporting the evolvability of such systems. In particular, NS theory requires that it must be possible to implement any added functionality with a fixed number of changes, in order to guarantee system-theoretic stability over time [2]. If implementing a new feature requires an unbounded set of changes, or a number of changes that depends on the size of the system, then this is referred to as a *combinatorial effect*. These effects are undesirable because, as the system grows over time, it will become too difficult and costly to change it. NS theory is a means to avoid combinatorial effects by design.

This is achieved by adopting a set of design principles and a software architecture that is based on the following elements:

- Data element – represents a data entity with primitive fields and possible references to other data entities.
- Action element – represents a functional task that consumes and produces data entities as input and output.
- Connector element – encapsulates I/O operations with external systems during the execution of an action.
- Workflow element – represents a sequence of actions which are called in a stateful way, by keeping state between consecutive calls.
- Trigger element – encapsulates the activation of an action or workflow in an automatic or periodic way.

NS theory ensures stability and evolvability with respect to a set of anticipated changes to these elements. Specifically, NS theory supports incremental changes such as creating additional Data elements, creating additional Action elements, adding Actions to Workflow elements, etc.[1] Through extensive use of the principle of version transparency (whereby an element can be replaced by a new version without breaking other elements), it can be shown formally that this set of anticipated changes can be performed on a normalized system without the occurrence of combinatorial effects [2].

A. Problem statement

One of the key elements in this 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, we argue that the implementation of process logic in this way actually goes against some of the fundamental principles on which NS theory is built, 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* between elements so that a change in one element has the least potential to require a change in another element.

As a composition of Actions, the Workflow element is a coarse-grained construct when compared to the other elements in the software architecture. It is also a factor of coupling between Actions, since these must bring the workflow from an initial state to a desired final state, with every state in between being both the output state of an Action and the input state of another Action. This creates a tightly coupled set of Actions which, in general, will be difficult to reuse in other workflows or even in different versions of the same workflow. If there is a change in the process logic, then this change may create a combinatorial effect that depends on the size of the workflow.

[1] In this paper we use capitalized words (e.g. Action) when we refer to specific elements in the software architecture of NS. We use non-capitalized words (e.g. action) to refer to general concepts beyond the context of NS.

B. Contribution

The goal of this work is to support changes in process logic while avoiding those combinatorial effects. For this purpose, we extend the software architecture of normalized systems with two complementary perspectives: the structural perspective, and the behavioral perspective. While the structural perspective is based on the usual Data and Action elements, the behavioral perspective introduces new elements, namely the Event element and the Rule element. As will be shown, Events and Rules are the behavioral counterpart of structural elements such as Data and Actions. The purpose of introducing these new elements is to create a fine-grained structure for implementing process behavior. This is illustrated by means of an application scenario in the retail industry, and we also discuss some details regarding its implementation.

C. Paper structure

The paper is organized as follows: Sections II and III discuss two important principles that justify the need for the behavioral perspective and also the use of events and rules. Section IV shows that these rules are conceptually similar to traditional ECA rules. Section V describes the proposed architecture, its software elements, the anticipated changes on those elements, and their run-time interactions. Section VI describes the application scenario in the retail industry where the proposed architecture supports several changes to a sales process. Finally, Section VII discusses the implementation of a new development framework for normalized systems based on the proposed architecture.

II. Duality of Structure and Behavior

In the late 1970s, Data Flow Diagrams (DFDs) [3], [4] became a popular tool for system analysis and design. Basically, DFDs described the data flows across system functions; in the Yourdon/DeMarco notation [3], 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 [5] 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 [6], although without settling on a standard language, as each workflow system used its own modeling language. The development of UML [7] 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 BPMN [8], which has some similarities to UML Activity Diagrams, but has been growing to address specific needs related to the modeling of business processes.

The point that we would like to make 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. In this work, we introduce Events and Rules in the behavioral perspective, as the dual elements of Data and Actions in the structural perspective.

III. Duality of States and Events

In the original NS theory [1], 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 1.

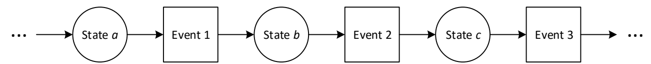

Fig. 1. States and events

The duality between states and events also exists in other process modeling languages. For example, in Petri nets [9] there are *places* and *transitions* which play a similar role to the states and events depicted in Figure 1 (even the notation is similar, since places are drawn as circles and transitions are usually drawn as thick bars or rectangles).

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 [10].

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, piecewise 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.

IV. ECA Rules

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 [11], WIDE [12], SWORDIES [13], and EVE [14]. Since then, several authors have proposed and discussed the benefits of using ECA rules for describing and implementing business processes [15]–[17]. 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.

Other common workflow patterns, such as those found in [18], 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 our proposed architecture, we 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.

V. Proposed Architecture

Figure 2 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 structure 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 structure for the implementation of process behavior.

The standard Connector and Trigger elements are also included, albeit in different perspectives. The Connector element supports the interaction with external systems; since Connectors are called by Actions, they have been included in the structural perspective. On the other hand, the Trigger captures events of interest from the outside; since Triggers

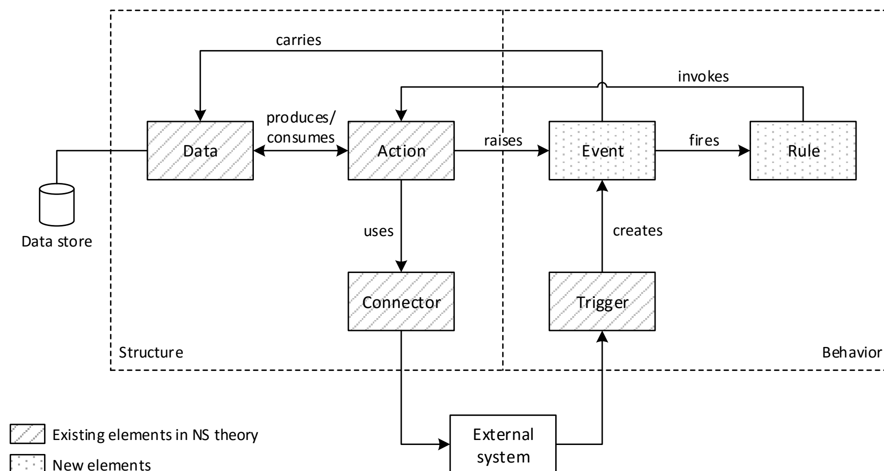

Fig. 2. Overview of the proposed architecture

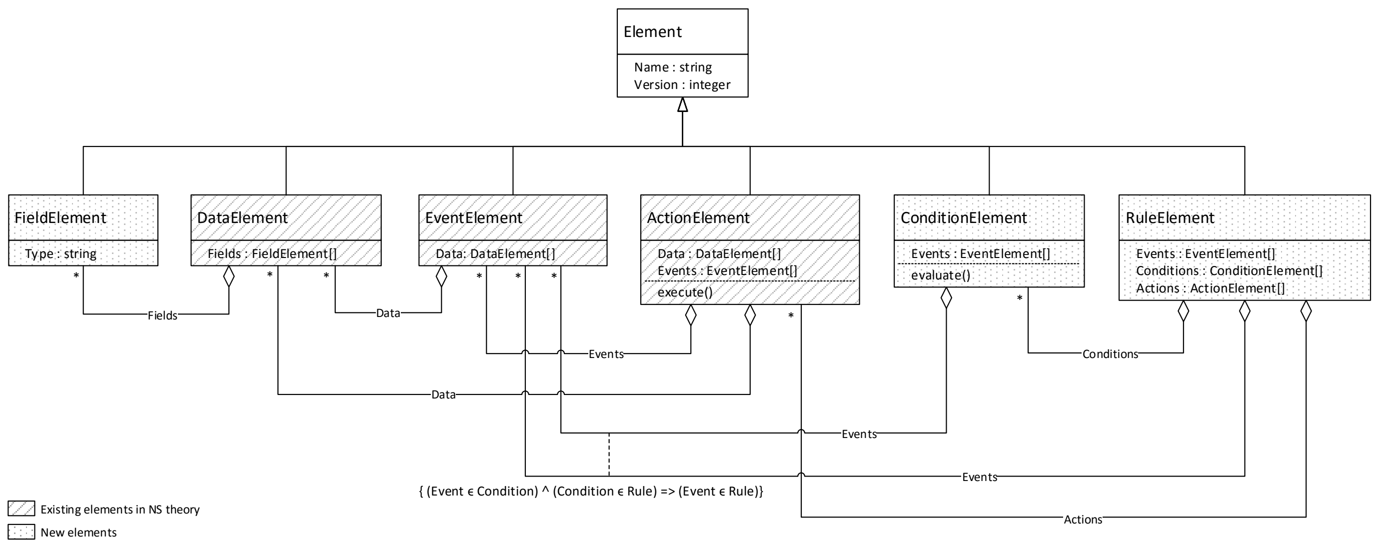

Fig. 3. Class diagram with inheritance and aggregation relationships

instantiate Events that fire Rules, they have been included in the behavioral perspective.

The way these elements interact with each other can be described as follows:

- *Actions manipulate Data*. Data elements can be both consumed and produced by Actions, as in standard NS theory. However, here we introduce 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.
- *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. Once the Event is raised, its payload can no longer be changed.
- *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.
- *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.
- *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.
- *Connectors and Triggers support external interactions*. Connectors support interactions initiated from the inside, and Triggers support interactions initiated from the outside. Asynchronous interaction patterns such as request-response and solicit-response [19] can be implemented through a combination of Triggers and Connectors.

Figure 3 provides a more detailed view of these elements and the relationships between them. For brevity, only inheritance and aggregation relationships are shown; Connector and Trigger elements have also been left out.

One of the fundamental principles of NS theory is version transparency [2], and this principle is supported here by having all elements descend from the base class Element, which provides a Name and Version.

A number of subclasses are derived from this base class. There are two additional elements that have been included to further refine this structure: one is the Field element, which is a component of the Data element, and the other is the Condition element, which is a building block of Rules.

In the Data element, the purpose of having Fields is to support changes in data types and also the addition/removal of Fields in a Data element, which are common changes in practice. In the Condition element, there is a list of Events to indicate which Events (actually, their Data) will be needed to evaluate the Condition. If a Rule has a Condition and that Condition needs certain Events, then these Events must belong to the set of firing Events of that Rule. We indicated this restriction with a constraint in the diagram.

For each element depicted in Figure 3, the list of anticipated changes is as follows:

- Field element – change of data type;
- Data element – addition and removal of Fields;
- Event element – addition and removal of Data;
- Action element – addition and removal of Data and Events;
- Condition element – addition and removal of Fields;
- Rule element – addition and removal of Events, Conditions and Actions.

From this description, it may appear that we support both the addition and removal of elements, while the original NS theory supports only additions and treats removals as a matter of garbage collection [20]. In fact, what we support here is the removal of a reference from a list or array of references. The element itself continues to exist while there are references to it. When the last reference is removed, it can be garbage collected, as advocated by standard NS theory.

VI. Application Scenario

To illustrate how the proposed architecture can be applied in practice, we turn to our 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.

A. *Sales process without promotions*

In the first stage of this case study, we focus on a sales process without any promotional campaigns. The goal is to identify the basic 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[2], 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 I shows these Events together with the Rules that they fire, the Actions invoked by those Rules, and the output Events raised by those Actions.

TABLE I
ELEMENTS FOR THE SALES PROCESS

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

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 4. The AddProductAction appends a new SalesLine to the Basket, and the AddPaymentAction sets the PaymentLine in the Basket.

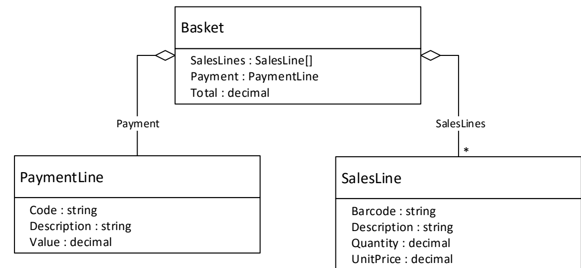

Fig. 4. Data elements for the sales basket (simplified)

B. *Sales process with direct discounts*

In the second stage of this case study, we modify the sales process 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, we need to introduce a new Data element to represent a direct promotion. In addition, 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.

The implementation of direct discounts requires the introduction of two new Rules. A first Rule checks whether there

[2]For an example of a POS system, see e.g.: https://www.toshibatec-ris.com/products_overseas/pos_system/. (TOSHIBA TEC Retail Information Systems is currently a leading provider of POS solutions after having acquired IBM's POS terminal business in 2012.)

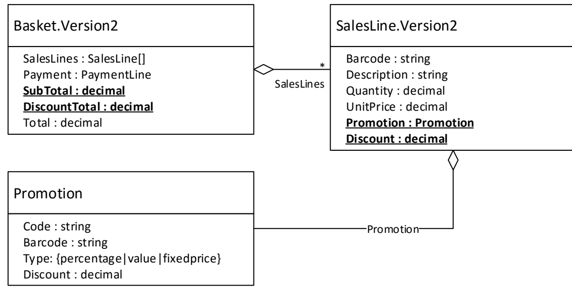

Fig. 5. Data elements for direct discount promotions

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 II shows the new elements to be added to the sales process.

TABLE II
ADDITIONAL ELEMENTS FOR DIRECT DISCOUNTS

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

C. *Sales process with "take x, pay y" promotions*

In the third stage of this case study, we focus on the possibility of having a different type of promotion which takes into account multiple sales lines in the basket. This type of promotions are usually called 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.

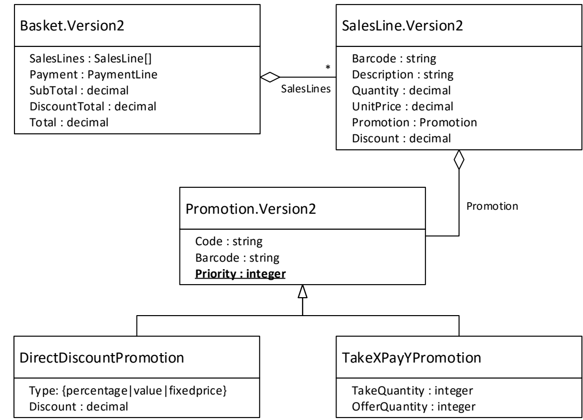

Fig. 6. Data elements for "take x, pay y" promotions

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 6.

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.

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 III shows the changes to the process.

TABLE III
CHANGES TO SUPPORT BASKET PROMOTIONS

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

VII. IMPLEMENTATION

Originally, the development framework for normalized systems was implemented over a J2EE platform based on Java technology and structured as a 4-Tier Web application architecture [1]. 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. Our approach in this work was to develop a prototype implementation of the proposed architecture by leveraging the rapid prototyping capabilities of the .NET Framework and Visual Studio.

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

- the .NET compilers and certain components of the .NET Framework have been recently open-sourced[3];
- it is possible to achieve a tight integration with Visual Studio through its SDK[4] and extensibility tools;
- it provides sophisticated code-generation capabilities through the use of T4 templates[5];
- in general, this platform and its tools offer great capabilities for rapid application development (RAD).

A. Defining the software elements

In our implementation, the software elements (Data, Actions, Events, Rules, etc.) are defined in an XML[6] file which is then validated against an XSD[7] schema definition. This XSD schema defines each element as a complexType that contains attributes and sequences of other elements. For example, the Data element extends a base type Element (which provides Name and Version), and contains a sequence of Field Elements. In essence, the XSD schema enforces the inheritance and aggregations relationships shown earlier in Figure 3.

A series of integrity constraints are also enforced after XSD validation. For example, one of such constraints is the one indicated in Figure 3: 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[8]. With LINQ, we write queries over object collections in order to verify that the element definitions are free of errors.

The process of translating the XML descriptor file into actual application code involves a series of stages as illustrated in Figure 7:

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) Semantic 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.

[3]Project "Roslyn" on GitHub.
[4]Software Development Kit (SDK)
[5]Text Template Transformation Toolkit (T4)
[6]Extensible Markup Language (XML)
[7]XML Schema Definition (XSD)
[8]Language-Integrated Query (LINQ)

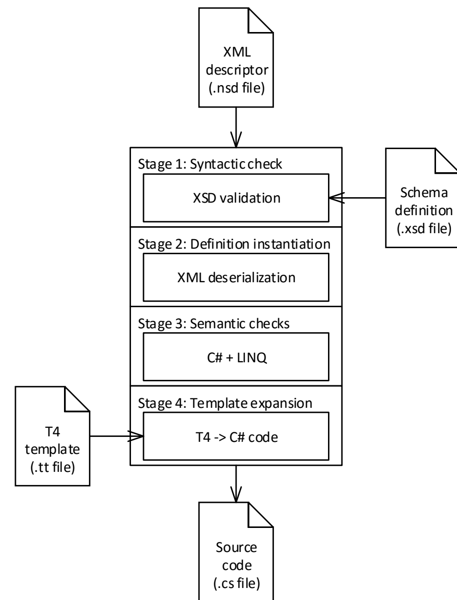

Fig. 7. Processing stages for code generation

B. Template expansion

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, we considered the option of using XSLT[9] to generate the source code from the XML descriptor file, but T4 revealed to be more adequate in this 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 source code.

In our T4 template, 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. Given the T4 template and a set of C# objects (the element definitions), the TextTemplatingFilePreprocessor generates the source code for the application based on the properties of those objects.

C. Custom tool

The four stages depicted in Figure 7 are implemented in a custom tool that we developed for Visual Studio. This custom tool is invoked whenever there is a change to the XML descriptor file. In this case, it re-generates the application source code automatically.

[9]Extensible Stylesheet Language Transformations (XSLT)

The custom tool was developed as a COM[10] 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.

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.

D. Visual Studio extensions

Although the software elements are to be defined in an XML descriptor file, we certainly do not intend to require the developer to write such XML file by hand. Therefore, we 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 8.

Basically, this custom designer is WPF[11] user control that uses a view model that is closely related to the Visual Studio project subsystem. 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.

Along with this extension, we also created a custom project template and a custom project item template. These templates appear in the "New Project" and "Add New Item" dialog boxes in Visual Studio, under a new "Normalized Systems" category that is present in the C# project/item template list.

[10]Component Object Model (COM)
[11]Windows Presentation Foundation (WPF)

E. Implementing version transparency

In the generated source code, we envisioned two ways to implement version transparency: one is through *inheritance* (where the new version is a subclass of the previous version) and another is through *containment* (where the new version contains an instance of the previous version).

A simple example is that of changing a Data element to include an additional Field. With inheritance, the new version of the Data element will extend the previous version with the additional Field, while still being able to behave as the previous version (its base class) for elements that have not been upgraded yet. With containment, the new class will contain a private member that is an instance of the previous version. In this case, it is possible to define a cast operator that returns the enclosed object, as follows:

```
public class Version2
{
   private Version1 _base;

   public Version2()
   {
      this._base = new Version1();
   }

   public static implicit operator Version1(Version2 obj)
   {
      return obj._base;
   }
}
```

This means that an object 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.

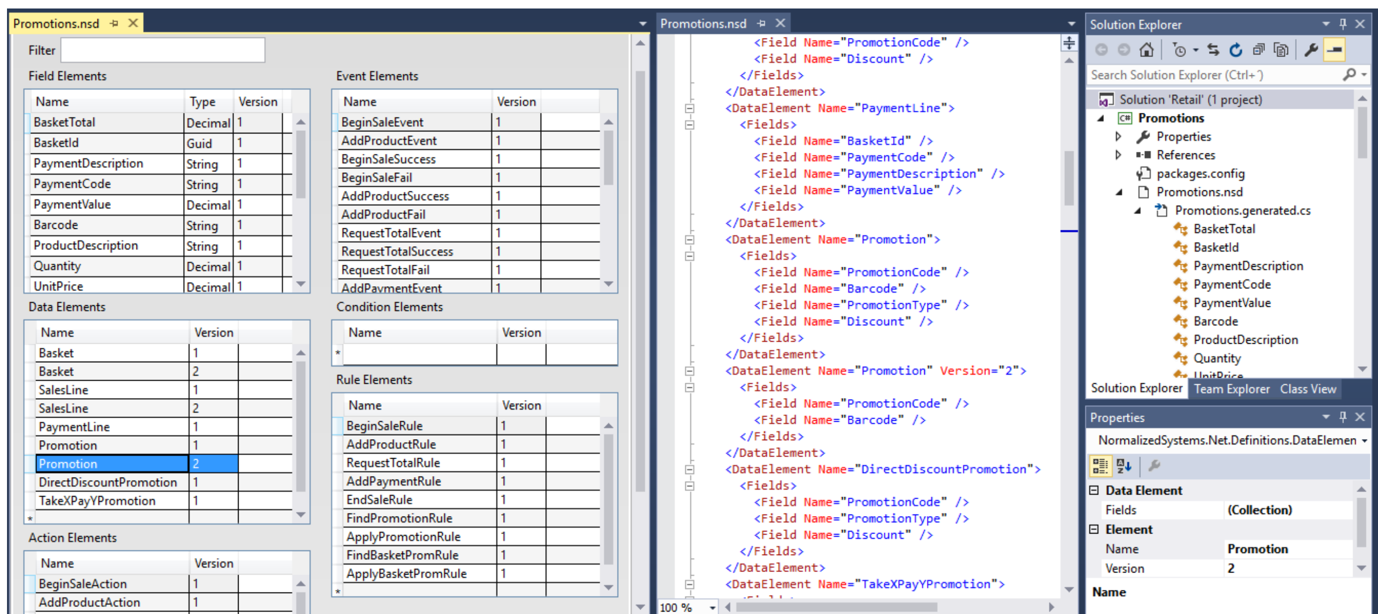

Fig. 8. Custom designer for creating an XML descriptor file

F. 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* [21] 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* [22], which consists in extracting any custom code from a previous version and re-inserting it into a new version of the application. In our 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.

G. Rule engine

The application code (with both the generated and the custom code) is compiled and linked to a library that provides common functionality for all target applications based on our implementation framework. An essential part of that common functionality is the *rule engine*, i.e. the component that manages the execution of the target application at run-time. This rule engine is based on an event queue on a rule instantiation mechanism, as depicted in Figure 9.

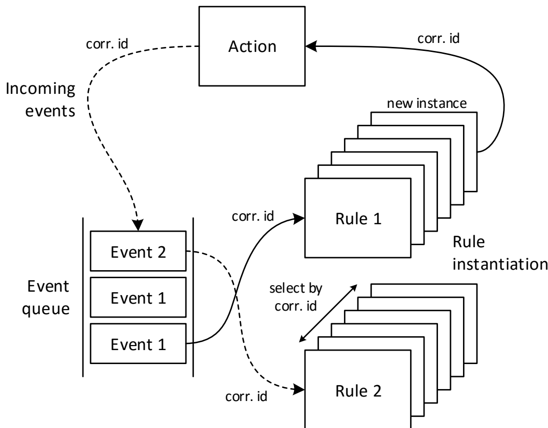

Fig. 9. Rule engine

First, we consider the scenario where each Rule is fired by a single Event. In this case, the rule engine retrieves an Event from the queue (in FIFO[12] order) and searches for Rules that are waiting for such Event. It then instantiates such Rules, evaluates their Conditions, and invokes their Actions if the Conditions yield true. 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). 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.

The use of correlation ids is an import mechanism in service interactions [23], where it is often necessary to associate an incoming message with a specific service instance. Here we use this mechanism 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.

VIII. Conclusion

In this paper we have proposed an improved architecture for Normalized Systems based on the idea of using ECA rules to implement process behavior. This is in contrast with current NS theory, which uses a graph-based approach to control process execution. 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, we observe 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.

[12]First in, first out (FIFO)

REFERENCES

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

[2] 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.

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

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

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

[6] 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.

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

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

[9] 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.

[10] R. Lu and S. Sadiq, "A survey of comparative business process modeling approaches," in *Business Information Systems*, ser. LNCS, vol. 4439. Springer, 2007, pp. 82–94.

[11] 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.

[12] 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.

[13] R. Endl, G. Knolmayer, and M. Pfahrer, "Modeling processes and workflows by business rules," in *1st European Workshop on Workflow and Process Management*. Zurich, Switzerland: Swiss Federal Institute of Technology (ETH), October 1998.

[14] A. Geppert and D. Tombros, "Event-based distributed workflow execution with EVE," in *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*. Springer, 1998, pp. 427–442.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] M. Brambilla, S. Ceri, M. Passamani, and A. Riccio, "Managing asynchronous Web services interactions," in *IEEE International Conference on Web Services*, 2004, pp. 80–87.

[20] 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.

[21] 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.

[22] 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.

[23] 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.