Realizing Business Processes with ECA Rules: Benefits, Challenges, Limits

François Bry, Michael Eckert, Paula-Lavinia Pătrânjan, and Inna Romanenko

Institute for Informatics, University of Munich
Oettingenstr. 67, D-80538 Munich, http://www.pms.ifi.lmu.de
{bry,eckert,patranjan}@pms.ifi.lmu.de, romanenk@cip.ifi.lmu.de

Abstract. Event-Condition-Action (ECA) rules offer a flexible, adaptive, and modular approach to realizing business processes. This article discusses the use of ECA rules for describing business processes in an executable manner. It investigates the benefits one hopes to derive from using ECA rules and presents the challenges in realizing business processes. These constitute a list of requirements for an (executable) business process description language, and we take them as a basis to investigate suitability of the concrete ECA rule language XChange in realizing a business process from the EU-Rent Case Study.

1 Introduction

Success in an increasingly global and competitive market requires companies to adjust internal activities and resources in an adequate and timely manner. Without suitable enterprise computing systems, this is infeasible. Managing and automating business processes is a key factor for successful enterprise computing systems.

A business process can be described as "a structured, measured set of activities designed to produce a specified output for a particular customer or market" [1]. Different methods and tools have been developed to describe business processes both for modeling purposes and for automatic execution. Such a description is also often called *workflow* or business *protocol*, and its automatic execution often called (workflow) *enactment*. The focus of this paper is on executable business process descriptions.

Recently, interest in rules is growing in different communities: companies manage and specify their business logic in the form of business rules [2], efforts are made for standardizing formats for rule interchange [3] as required for example in policy-based trust negotiations [4], and rule languages are becoming popular for reasoning with Web and Semantic Web data [5]. Like rules in general, Event-Condition-Action rules offer a flexible, adaptive, and modular approach to realizing business processes.

In this article we analyze realizing business processes (i.e., describing business processes in an executable manner) based on ECA rules. The focus is on control flow, because this is the aspect one is most concerned about during specification; other issues are shortly discussed.

We investigate the benefits one hopes to derive from using ECA rules (Section 2) and present the challenges in realizing business processes (Section 3). These constitute a list of requirements for an (executable) business process description language, and we take them as a basis to investigate suitability of the concrete ECA rule language XChange in realizing a business process from the EU-Rent Case Study (Section 4). We close with a discussion of the practical limits of ECA rules for business processes (Section 5) and conclusions (Section 6).

2 ECA Rules for Business Processes: Benefits

Managing and automating a business process requires a machine-readable description of the business process. The most widely used language for describing business process today is the Business Process Execution Language (BPEL) [6].

Simplified, BPEL describes a process as activities (typically provided as Web Services) with control flow (e.g., sequential execution) in an imperative fashion. Additionally, handlers to catch errors or other exceptional situations in the process can be specified.

In this article we argue for a different approach to describing business processes based on ECA rules. ECA rules have the form "on *event* if *condition* do *action*" and specify to execute the action automatically when the event happens, provided the condition holds. Whereas traditional business process description languages center around activities, ECA rules put emphasis on events. An ECA-rule-based approach for specifying business processes can have the following advantages:

- Requirements are frequently specified in the form of rules expressed in either a natural or formal language, in particular business rules, legislative rules, or contractual rules. In requirements on business processes, we often find ECA rules such as "a credit card application *(event)* will be granted *(action)* if the applicant has a monthly income of more than EUR 1.500 and no outstanding debts *(condition)*." Ideally, a one-to-one mapping between rules used for requirements specifications and (executable) rules used for workflow enactment can be achieved.
- Reactive rules, especially ECA rules, easily integrate with other kinds of rules commonly used in business applications such as deductive rules (rules expressing views over data or rules used for reasoning with data) and normative rules (rules expressing conditions that data must fulfill; also called integrity constraints). Methods for automatic verification and validation of rule sets have been well-studied in the past and can be applied.
- ECA rules have a flexible nature: they are easy to adapt, alter, and maintain as requirements change, which is quite frequently the case for business processes. Even more, many rule engines allow rules to be added, modified, or deleted "on-the-fly," i.e., without interrupting running processes.
- An important part of business process descriptions is handling of errors and exceptional situations; in fact, it is often the longest and most labor-intensive

part. Since errors and exceptional situations can be conveniently expressed as (special) events, ECA rules allow to treat them just like "normal" situations, thus making their handling quite easy.
- Rules can be managed in a single rule base as well as distributed in several rule bases. The latter is advantageous for cross-enterprise processes, where there is no central instance (such as a workflow management system) executing and monitoring processes.
- In an activity-centered control flow, activities are started as reaction to the (successful or unsuccessful) completion of another activity; reaction to intermediate states of activities are typically not supported [7]. ECA rules with their emphasis on events offer more flexible means to specify control flow, if appropriate events are generated by the activities.

Whether an activity-centered or an event-centered approach for describing business processes is better suited depends, of course, always on the individual process and its environment. In situations where modeling and specifying a process is better done with an activity-centered view, it is usually possible to automatically or semi-automatically derive ECA rules realizing the execution of the process.

3 Challenges for Realizing Business Processes

Every business process execution language should answer certain requirements for effective and efficient support of business processes, primarily the ability to realize separate activities (tasks or steps) and to control their cooperation (or interworking). In this section we present the essential challenges for realizing business processes.

3.1 Control Flow

Control structures are the core elements of every business process modeling (or execution) language. They describe temporal and logical dependencies between activities such as: sequential execution of activities, parallel execution, synchronization, alternative execution. Van der Aalst et al. [8] have identified 21 patterns of control flow ranging from the simple patterns just named to more complex, process specific patterns. The technical ability of a business process description language to express these patterns can be viewed as an essential indicator of the language usability to design and implement business processes.

Consider the business process for handling a rental reservation (RR) depicted in Figure 1 in Business Process Modeling Notation (BPMN) [9]. A customer invokes the business process by sending a rental reservation request to a rental company.

If the customer is already registered, the customer blacklist is checked (*check blacklist*); in case the customer is on the blacklist, the rental request is rejected (*send rejection to customer*) and the process ends. If the customer is not registered yet, her data are recorded (*introduce new customer*).

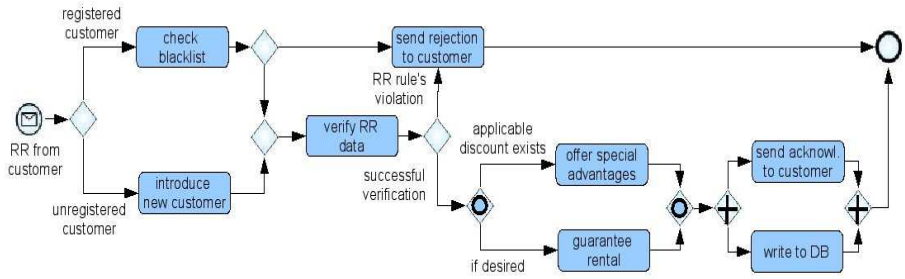

Fig. 1. Reservation business process represented in BPMN

In the next step the reservation data is checked (*verify RR data*); e.g., that there are no overlaps with other reservations of the customer and that a car in the specified group is available. In case of a violation, the process ends again with a rental rejection.

Next, a number of activities are performed, which depend on certain conditions (possibly no activity, if none of the conditions holds). If an applicable discount exists, it is offered to the customer (*offer special advantages*). If the customer's rental request indicates that a guaranteed rental (car can be picked up within 24 hours after the scheduled pick-up time) is desired, corresponding arrangements are made (*guarantee rental*).

Finally, the customer is notified that her rental request has been accepted (*send acknowl. to customer*) and, in parallel, the rental reservation is recorded (*write to DB*). With this the process ends.

We will now analyze the control flow patterns in this example:

- **Sequence** is the most basic control pattern; it runs two (or more) activities one after the other. In the example, *introduce new customer* and *verify RR data* are in a sequence. Sequencing of activities is drawn as a solid arrow in BPMN.
- **Exclusive Choice** allows execution of exactly one alternative path chosen at the runtime based on the evaluation of a condition. In the example, one of *check blacklist* and *introduce new customer* is chosen exclusively, based on whether the customer is registered. The alternative execution paths can be brought together again with a **Simple Merge**; when the chosen activity of the Exclusive Choice finishes, execution continues with the activity after the corresponding Simple Merge. Both patterns are represented in the BPMN with simple diamonds (so-called XOR gateways).
- **Multi-Choice** is similar to Exclusive Choice, but allows *more than one* alternative paths to be chosen and executed in parallel, or even to execute no path at all. The counterpart to join the parallel execution paths (or continue execution if no path has been chosen) is the **Synchronizing Merge**, which waits for all chosen paths to finish before continuing. BPMN uses diamonds with circles inside (OR gateways) to depict these patterns. In our example,

either both *offer special advantages* and *guarantee rental*, only one of them, or none are to be executed, depending on the stated conditions.
- **Parallel Split** executes multiple paths (with independent activities) in parallel. **Synchronization** joins them again by waiting for all paths to finish. In the graphical representation both patterns are indicated through diamonds with plus symbol (AND gateways). In the example, *send acknowl. to customer* and *write to DB* are independent activities performed in parallel.

There are more control flow patterns, but the above are the most common and are supported by virtually all business process modeling or execution languages. The realization of the above patterns with ECA rules will be investigated in the next section.

3.2 Process Instances

Another challenge for a business process language is the ability to support performing of different activities within a *process instance*. A process instance is the execution representation of a process. Considering the example of handling a rental request: a process instance is created each time when a rental request from a customer is received. Several process instances corresponding to different rental requests (possibly from the same customer) run in parallel.

When a cancellation request from a customer arrives, this is specific to one of the customer's previous rental requests. It should cancel only the one corresponding process instance, not all processes. A business process language hence must provide a mechanism that assigns events that happen as well as running activities to their corresponding process instances.

3.3 Integration with Business Rules

Business rules are used for defining or constraining aspects of business, such as inserting business structure or controlling or influencing the behavior of business. They represent the business logic of a company and exist in every enterprise. Often the logic of workflow-based systems is given or influenced by business rules. Frequently the rules are embedded within the business process itself which makes changing and maintaining business rules difficult and costly. Recently business rules management, i.e., separating business processes and business rules, and formally specifying, enforcing, integrating, and maintaining business rule sets, has gained much attention.

Business rules can be classified according to their effect. A common classification [10] distinguishes three types:[1]

- structural rules (also called normative rules or constraints) define restrictions on business concepts and facts,

[1] Other classifications for business rules [11] or rules in general [12] exist; the presented classification is well-accepted, clear, and suitable in the framework of this article.

– derivation rules (also called deductive or constructive rules) are statements of knowledge derived from other knowledge using inference or mathematical calculations,
– dynamic rules (also called active, reactive, or reaction rules) concern dynamic aspects of the business; they constrain or control the actions of business.

In business processes, business rules play an important role at decision points, where processes change their behavior based on certain criteria or rules. The most common approaches for integrating business rules and business processes are (1) checking the rules explicitly as activities, e.g., calling a rule engine Web Service, and (2) checking the rules implicitly at decision making points. A business process language should support the integration of business rules.

3.4 Exception Handling

The ability to specify exceptional conditions and their consequences, including recovery measures, are as important for realizing business processes as the ability to define "normal behavior." An exceptional situation in the process of Figure 1 could, for example, occur during the *verify RR data* activity if the credit card of a customer has expired. Possible means for recovery include asking the customer for updated information or canceling the whole rental request.

Because of the multitude and diversity of exceptional situations, the effort for specifying exception handling often surpasses the effort for specifying normal behavior. Hence, every business process language should provide a systematic and elegant mechanism to specify, handle, and recover from exceptions.

3.5 Abstractions for Reusability and Maintainability

Business process specifications should exhibit modular structure to ease reusing and maintaining parts of the specifications such as sub-workflows. An important step towards reusability and maintainability is the integration of business rules into process specifications (see above). However, further means are required: for example, use of a sub-workflow in several other workflows requires support from the business process language and cannot be realized through the integration of business rules.

Modularity is best explored in object-oriented and procedural languages; for rule-based languages, however, modularity is still a research challenge.

4 Realization in XChange

In this section we demonstrate the capabilities of the rule-based language XChange in realizing business processes. The concrete processes go along the lines of the EU-Rent Case Study [13], a specification of business requirements for a fictive car rental company, promoted by the European Business Rules Conference [14] and the Business Rules Group [15]. Our focus is on control flow, but we also discuss the integration of business rules and other issues touched on in Section 3.

```
<xchange:event>
  <reservation-request>
    <customer>John Q Public</customer>
    <email>john@public.com</email>
    <car>
      <group> A </group>
    </car>
    <period>
      <from>2006-06-10</from>
      <duration> 2 </duration>
    </period>
    <location>Budva</location>
  </reservation-request>
</xchange:event>
```

```
xchange:event [
  reservation-request {
    customer { "John Q Public" },
    email { "john@public.com" },
    car {
      group { "A" }
    },
    period {
      from { "2006-06-10" },
      duration { "2" }
    }
    location { Budva }
  }
]
```

Fig. 2. Reservation Request (RR) event in XML and term representation

```
ON    xchange:event {{
        reservation-request [[
          var Customer -> customer {{ }} ]]  }}
FROM in { resource {"http://rent.eu/customers.xml" },
        customers {{
            without var Customer }} }
DO    in { resource {"http://rent.eu/customers.xml" },
        customers {{
            insert var Customer }} }
END
```

Fig. 3. XChange ECA rule for introducing a new customer

4.1 XChange in a Nutshell

XChange is a reactive language based on ECA rules and is tailored to the Web and XML data, which makes it an interesting candidate for realizing and composing Web Services. An XChange program is located at one Web node and consists of one or more (re)active rules of the form *event query — condition query — action*. Events are represented and communicated between different Web nodes as XML messages (e.g., with SOAP [16]). Every incoming event is queried using the event query (introduced by keyword `ON`). If an answer is found and the condition query (introduced by keyword `FROM`), which can query arbitrary Web resources, has also an answer, then the specified action (introduced by keyword `DO`) is executed.

Event queries, condition queries and actions follow the same approach of specifying patterns for the data that is queried, updated, or constructed. XChange embeds the XML query language Xcerpt [17] and extends it with update facilities and reactivity.

The parts of an XChange ECA rule communicate through variable substitutions. Substitutions obtained by evaluating the event query can be used in the condition query and the action part, those obtained by evaluating the condition query can be used in the action part.

Example. Figure 2 depicts an incoming rental request event. On the left it is in XML syntax, on the right it is in XChange's term syntax, which is used for conciseness in data, queries, and updates. Figure 3 depicts an XChange ECA

rule which reacts to this event (ON-part), checks that the customer is not yet registered (FROM-part), and inserts him into the customer database (DO-part).

In the term syntax, square brackets `[ ]` denote that the order of the children of an XML element is relevant, curly braces { } denote that the order is not relevant.

In event queries and condition queries, both partial (i.e., incomplete) or total (i.e., complete) query patterns can be specified. A query term t using a partial specification (denoted by double brackets or braces) for its subterms matches with all such terms that (1) contain matching subterms for all subterms of t and that (2) might contain further subterms without corresponding subterms in t. In contrast, a query term t using a total specification (denoted by single brackets or braces) does not match with terms that contain additional subterms without corresponding subterms in t. Query terms contain variables for selecting subterms of data terms that are bound to the variables. Using "`->`" (read "as"), a restriction can be made on the bindings of the variable left of "`->`"; every binding has to match the (sub-)query to the right. The results of a query are bindings for the free variables in that query. In the example, `Customer` is bound to `customer { "John Q Public" }`.

Updates in the action part are queries to Web resources, augmented with the desired update operations (`insert`, `delete`, `replace-by`). Another form of action supported by XChange is the raising of a new event.

XChange is a rich language and we will discuss further constructs relevant in the scope of this article as we go along. For a short introduction to XChange see [18], for a complete introduction accompanied by the specification of declarative and operational semantics see [19].

4.2 Control Flow

We start off by implementing the control flow for the process from Figure 1 by refining the rule from Figure 3. The first control flow pattern in the process is the Exclusive Choice: the next action depends on the condition of being a registered customer. Such a choice is conveniently implemented by means of an extended form of ECA rules, called ECAA rule [20]: the event is the *reservation request* (message in Figure 2), the condition *customer unregistered* (a query to a database), the action to be executed is either *introduce new customer* (in case the condition holds) or *check blacklist* (otherwise).

ECAA rules are only syntactic sugar that significantly increases readability of rule sets; every such ECAA rule can be translated into two ECA rules with one condition being the negation of the other.

Figure 4 shows an XChange ECAA rule implementing the Exclusive Choice. The action in case the customer is unregistered (DO-branch) is the update to the customer database already discussed. To continue in the process after this action, an event is raised. This event will trigger further rules implementing part of the whole process. This style of implementing the sequence pattern by passing along events is common for ECA rules [20].

```
ON   xchange:event {{
         var Rental -> reservation-request {{
             var Customer -> customer {{ }}  }}  }}
FROM in { resource {"http://rent.eu/customers.xml"},
          customers {{ without var Customer }} }
DO   and [
       in { resource {"http://rent.eu/customers.xml" },
            customers {{ insert var Customer }}
       },
       xchange:event {
          new-customer { var Rental }   }  ]
ELSE xchange:event {
         check-blacklist { var Rental } }
END
```

Fig. 4. Exclusive choice on the customer's registration status

```
ON xchange:event {{
       blacklisted { var Rental }  }}
DO reply-customer [ var Rental, "Failure", "You are blacklisted." ]
END
```

Fig. 5. Send rejection to customer if blacklisted

For the ELSE-branch, the action for *check blacklist* is simply sending a message to a Web Service implementing it. The answer from the service is another message that will (just as the event raised in the DO-branch) trigger further rules.

The rule in Figure 5 implements the reaction to a positive answer from the *check blacklist* service. The action *send rejection to customer* is implemented as a procedure and will be discussed later.

Next, the process merges a negative answer from *check blacklist* and an answer from *introduce new customer*, and continues with the *verify RR data* action. The corresponding rule in Figure 6 uses a disjunction of events to implement the Simple Merge and raises an event that is sent to a service taking care of testing compliance of the rental request with the company's business rules.

In case the rental request satisfies the rental rules, *verify RR data* replies with an event `RR-ok`; otherwise with `RR-not-ok` (which contains a reason for rejection). The rule in Figure 7 reacts to this `RR-not-ok` event and sends a rejection message to the customer (in analogy to the rule for blacklisted customers in Figure 5).

```
ON or {
     xchange:event {{ not-blacklisted { var Rental } }},
     xchange:event {{ new-customer { var Rental } }}     }
DO xchange:event { verify-RR-rules { var Rental } }
END
```

Fig. 6. Simple Merge of the *check blacklist* and *introduce new customer* branches

```
ON    xchange:event {{
          RR-not-ok { var Rental -> reservation-request {{ }},
                      var Message -> message {{ }} }  }}
DO    reply-customer [ var Rental, "Failure", var Message ]
END
```

Fig. 7. Send rejection to customer if RR rules are violated

```
ON    xchange:event {{
          RR-ok { var Rental -> reservation-request {{ guarantee {"yes"} }},
                  var Price -> price {{ }} }  }}
DO    xchange:event {
          guarantee-rental [ var Rental, var Price ] }
END

ON    xchange:event {{
          RR-ok { var Rental -> reservation-request {{ guarantee {"no"} }},
                  var Price -> price {{ }} }  }}
DO    xchange:event {
          guarantee-done [ var Rental, var Price ] }
END

ON    xchange:event {{
          RR-ok { var Rental -> reservation-request {{
                      period { from { var From }, duration { var Duration } },
                      car {{ group { var Group } }}  }},
                  var Price -> price {{ }} }  }}
FROM  exist-discounts [ var From, var Duration, var Discount ]
DO    xchange:event {
          apply-discounts [ var Rental, all var Discount, var Price ] }
ELSE  xchange:event {
          discounts-done [ var Rental, var Price ] }
END
```

Fig. 8. Multi-Choice for *offer special advantages* and *guarantee rental*

The rules in Figure 8 implement the Multi-Choice in the process following successful verification of the rental rules (event `RR-ok`). The corresponding Synchronizing Merge is implemented in the event part of the rule in Figure 9; it uses a conjunction of events to merge. Note that the events `guarantee-done` and `discounts-done` are generated whether the corresponding actions are executed or not. This is necessary for the merge.

The upcoming Parallel Split is also implemented in the rule in Figure 9, namely in the action part. With the actions *send acknowl. to customer* and *write to DB* the process ends. For simplicity, we skipped the Synchronization of the action before the end of the process. It could be implemented in the same manner as the Synchronizing Merge.

4.3 Abstractions and Business Rules Integration

Reusability and maintainability of business processes can be greatly increased by convenient abstraction mechanisms and the integration of business rules.

An important abstraction mechanism is the capability to bundle actions that are complex or used frequently into procedures [21]. An action used frequently

```
ON and {
      xchange:event {{
          guarantee-done {{
              var Rental -> reservation-request {{ var Var }} }} }},
      xchange:event {{
          discounts-done [
              var Rental, var Price ]  }}  }
DO and {
      in { resource {"http://rent.eu/rentals.xml" },
          rentals {{ insert rental { all var Var, var Price } }} },
      reply-customer { var Rental, "Finished", "Reservation successful." }
END
```

Fig. 9. Synchronizing Merge of the above Multi-Choice and Parallel Split for *send acknowl. to customer* and *write to DB*

```
PROCEDURE reply-customer [
                reservation-request {{ email { var Email } }},
                var Status, var Message ]
DO          xchange:event {
                xchange:recipient { var Email },
                eu-rent-reply [ var Status, var Message ]  }
END
```

Fig. 10. Procedure for sending a reply to the customer

in our process is the reply to a customer. Figure 10 demonstrates defining such a procedure in XChange. It is called in the rules of Figures 5, 7, and 9.

Concerning the integration of business rules, the *verify RR rules* action illustrates checking business rules explicitly as activities by calling some service. This service can actually be implemented in XChange, see Figure 11. The implementation consists of one ECA rule reacting to the incoming event `verify-RR-rules`, a deductive rule for `get-price` (the result of which is queried in the condition part of the ECA rule), and two procedures `car-unavailable` and `rental-overlaps` for checking their corresponding business rules. (For space reasons, the figure only shows the procedure `car-unavailable`; the other procedure is similar.) The procedures generate appropriate replies in case of violations; if no violation is detected, the `RR-ok` event is raised by the ECA rule.

4.4 Process Instances

To deal with process instances in ECA rules different approaches are conceivable:

- `Rule tied to process instances:` In this approach, rules are executed as part of a process instance; their event queries only see events that are part of this process. Special constructs are required to start and end processes, and fork and join sub-processes.
- `Rule outside process instances:` In this approach, rules run separated from process instances; events have to carry some identifier for the process they belong to. This identifier has to be explicitly queried in the event part of rules and passed on by the actions and used services.

```
ON    verify-RR-rules {
          var Rental -> reservation-request {{
              car {{ group { var Group} }},
              period { from { var From }, duration { var Duration } }
}} }
FROM get-price [ var Group, var From, var Duration, var Price ]
DO    or [
          car-unavailable { var Rental },
          rental-overlaps { var Rental },
          xchange:event { RR-ok [ var Rental, var Price ] }  ]
END

CONSTRUCT get-price [ var Group, var From, var Duration, var Price ]
FROM      in { resource {"http://rent.eu/prices.xml" },
               desc car-group {{
                   name { var Group },
                   prices {{
                       rental {
                           duration { var Duration },
                           price { var Price } } }}  }} }
END

PROCEDURE car-unavailable {
               var Rental -> reservation-request {{
                   car {{ group { var Group } }}  }}  }
FROM      in { resource {"http://rent.eu/rentals.xml" },
               without desc car {{
                   model { var Group },
                   car-status { "available" }  }}  }
DO        xchange:event {
              RR-not-ok {
                   var Rental,
                   message {"Selected car group unavailable"} } }
END
```

Fig. 11. Implementation of the *verify RR data* activity in XChange

In the presented rules the second approach has been used, using as identifier of the process instance the `rental-request`-information, which is passed along through all events.

The disadvantage of the approach with rules outside process instances is that it puts more burden on the programmer's shoulders: the rules have to query the identifier in the event part (e.g., to ensure that for a conjunction of events only events of the same process instance are used) and passed along in every event that is raised. Approaches where rules are tied to process instances are hence more convenient.

However, tying rules to process instances is sometimes not possible for distributed workflows, in particular cross-enterprise workflows or workflows depending on events from other, parallel workflows.

4.5 Exception Handling

Since exceptions can be conveniently expressed as (special) events, ECA rules are a convenient mechanism for handling exceptions. They allow to treat exceptions like any other event.

The process we presented did not contain any exceptions as such, though one could argue that a customer being blacklisted or a violation of the rental rules could be perceived as an exception.

Exception handling is not a focus in this paper; however, ECA rules have quite successfully been employed for exception handling in the past [22].

5 ECA Rules for Business Processes: Limits

While using ECA rules in realizing business processes has the benefits outlined in Section 2, the approach has also some practical limits, just like any other approach. In the following we present the limits we have identified from the concrete study of using the ECA rule language XChange for specifying executable business processes presented in Section 4.

A general limit of ECA rules is that they do not always reflect the procedural, imperative way of thinking familiar to many people from imperative or object-oriented programming. This is particularly obvious when looking at the realization of the sequence pattern with ECA rules: for sequencing of activities A and B, B is triggered by a separate rule which reacts on a finish event of A. (XChange alleviates this to some degree through the `and[...]` when the activities are updates, though this is only a special case.) However, for distributed workflows without a central coordinator (e.g., cross-enterprise workflows) this style of programming is not unnatural and hard to avoid.

Closely related to this is that ECA rules usually do not have a local state that is specific and internal to the current process instance. ECA rules have to explicitly maintain this state in events and databases. For example, an incoming rental request could contain information such as the customer's e-mail address. This information is not needed immediately in the process of Figure 1, but only late in the process for sending rejection or acknowledgment back. In the first rule, this information has to be either saved in a database or passed along through all rules as part of event data.

Monitoring of business processes specified with ECA rules is not as straightforward as for other approaches, which are based on some activity-centered, automata-like model (e.g., BPMN or BPEL). In activity-centered process specifications, a process's state is obvious from the finished and running activities. In contrast, in event-centered process specifications, the process's state is given through the history of events, which is less easy to comprehend.

Because of this "hidden" process state in ECA rule-based specifications, there is no clear notion of which events are expected next. (Enabled) ECA rules are triggered by every incoming event matching the event query, regardless of whether this event is expected or not. This might entail unexpected behavior, especially if events are generated "out-of-order" by faulty or malicious behavior of systems. Possible solutions to this depend on the ECA rule language's capabilities. Dynamic enabling and disabling of rules provides an approach on a meta-level; however an activity-centered solution is much simpler.

Current rule languages have only limited support for structuring rule sets. In practice, however, structuring is very much needed to reduce the complexity and expenses in the production of business process specifications. It can be expected that this issue will be overcome through further research and practical experiences, in particular by adding support for modules.

Last not least, development and maintenance of business processes is greatly supported by visual tools. For event-centered approaches, visualizing single ECA rules alone does not suffice: it is important to visualize whole rule sets with the associated control flow. Again, this issue might be overcome through further development.

Of these limits, the first four stem from the rule-based, as opposed to an imperative, programming style. How strongly this limits the applicability of ECA rules to business processes hence also depends largely on the experience with rule-based programming. The latter two limits are more a limitation of *current* rule languages and expected to be solved in the near future.

6 Conclusion and Outlook

In this article we have analyzed the realization of business processes by means of ECA rules. With a focus on control flow, we have presented an implementation of a concrete business process scenario in XChange. This work has greatly influenced and advanced the development of XChange as a reactive language. In particular, it has led to the introduction of a procedure notion, which is absent in most other rule languages. Constructs for structuring rule sets in XChange are an issue of ongoing development deserving refinement and further research.

Issues also deserving attention for future work regard exception handling in connection with transactions and compensating actions, as well as issues relating to process instances.

As this paper has shown, there is still a lot to be done for using ECA rules in business processes. The first results of this paper are promising and give requirements and guidelines for future work, in particular on language design.

Acknowledgments

This research has been funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (http://rewerse.net).

References

1. Davenport, T.H.: Process Innovation: Reengineering Work through Information Technology. Havard Business School Press (1993)
2. The Business Rules Group: Defining business rules – what are they really? Available at www.businessrulesgroup.org (2000)

3. World Wide Web Consortium: Rule interchange format working group charter. See www.w3.org/2005/rules/wg/charter (2005)
4. Bonatti, P.A., Olmedilla, D.: Driving and monitoring provisional trust negotiation with metapolicies. In: IEEE Int. Workshop on Policies for Distributed Systems and Networks, IEEE Comp. Soc. (2005)
5. Bry, F., Schwertel, U.: REWERSE – reasoning on the Web. AgentLink News (15) (2004)
6. Andrews, T., et al.: Business process execution language for web services version 1.1. Available at www.ibm.com/developerworks/library/ws-bpel (2003)
7. Carter, B.M., Lin, J.Y.C., Orlowska, M.E.: Customizing internal activity behaviour for flexible process enforcement. In: Proc. Australasian Database Conference, Australian Computer Society (2004)
8. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distributed and Parallel Databases **14**(1) (2003)
9. White, S.A.: Introduction to BPMN. Technical report, Object Management Group (OMG) (2004) Available at www.bpmn.org.
10. Hall, J.: Business rules boot camp. Tutorial at the European Business Rules Conference (2005)
11. Wagner, G.: How to design a general rule markup language? In: Proc. Workshop on XML Technologien für das Semantic Web - XSW. Volume 14 of LNI, GI (2002)
12. Bry, F., Marchiori, M.: Ten theses on logic languages for the Semantic Web. In: Proc. Int. Workshop on Principles and Practice of Semantic Web Reasoning. Volume 3703 of LNCS, Springer (2005)
13. EU-Rent Case Study. www.eurobizrules.org/ebrc2005/eurentcs/eurent.htm (2005)
14. European Business Rules Conference. www.eurobizrules.org (2005)
15. Business Rules Group. www.businessrulesgroup.org (2005)
16. Gudgin, M., et al.: SOAP version 1.2. W3C recommendation, World Wide Web Consortium (2003)
17. Schaffert, S., Bry, F.: Querying the Web reconsidered: A practical introduction to Xcerpt. In: Proc. Extreme Markup Languages. (2004)
18. Bailey, J., Bry, F., Eckert, M., Pătrânjan, P.L.: Flavours of XChange, a rule-based reactive language for the (Semantic) Web. In: Proc. Int. Conf. on Rules and Rule Markup Languages for the Semantic Web. Volume 3791 of LNCS, Springer (2005)
19. Bry, F., Eckert, M., Pătrânjan, P.L.: Reactivity on the Web: Paradigms and applications of the language XChange. J. of Web Engineering **5**(1) (2006) 3–24
20. Knolmayer, G., Endl, R., Pfahrer, M.: Modeling processes and workflows by business rules. In: Business Process Management. Volume 1806 of LNCS, Springer (2000)
21. Bry, F., Eckert, M.: Twelve theses on reactive rules for the Web. In: Proc. Workshop Reactivity on the Web at Int. Conf. on Extending Database Technology. Volume 3268 of LNCS, Springer (2006)
22. Brambilla, M., Ceri, S., Comai, S., Tziviskou, C.: Exception handling in workflow-driven web applications. In: Proc. Int. Conference on World Wide Web, ACM (2005)