Dependencies in Business Process Rule Hierarchies

Elke Pulvermüller
Department of Mathematics and Computer Science
University of Osnabück
D-49076 Osnabrück, Germany
Email: elke.pulvermueller@informatik.uni-osnabrueck.de

Andreas Speck, Sven Feja and Sören Witt
Institute for Computer Science
Christian-Albrechts-University of Kiel
D-24098 Kiel, Germany
Email: {aspe|svfe|swi}@informatik.uni-kiel.de

Abstract

Automated checking concepts for business process models support human testers considerably by saving time. However, this new checking ability results in a comparatively large number of rules representing requirements. But without a comprehensible representation of the relations between the rules on the one hand it’s hard to keep track on the validated rules and on the other hand to correctly interpret the validation results.

In this paper we propose an improvement for the automated validation of business process models by offering elements to create abstract rules and arranging these rules in hierarchies. Top-down and bottom-up testing are supported by stepwise activating (and validating) the rules starting from the top of the hierarchy (or bottom respectively). Moreover, the rule hierarchies may be reused when similar systems are to be validated by configuring a valid rule sub-set for the specific business process system.

I. INTRODUCTION

Business process models are used to describe the behaviour of commercial information systems. Different modelling concepts have been established like the Business Process Model and Notation (BPMN [1]) or the Architecture of integrated Information Systems modelling concept (ARIS [2]). In our paper we use the Event-driven Process Chain (EPC) of ARIS to model business processes. We made this choice since the application example from the data protection domain of the paper is modelled in EPC [3], [4].

Rules express requirements which must be fulfilled by the business information systems. These rules may consider the static dependencies in a system and the dynamic behaviour of its processes. Manual testing is one possibility to assure that the systems are compliant to the rules. However, in order to optimize the quality assurance (save time and less errors caused by humans) a further automated checking support is desirable. There are already prototypes and research systems providing checking mechanisms. In our work we use the automated checking system Business Application Modeller (BAM) (we present this system amongst others in the related work section II). We realised that already comparatively small business process driven systems are defined by a large number of rules representing the requirements.

In section III we introduce the business process model notation EPC and present a small example process. Furthermore, we present a graphical notation to express temporal rules (G-CTL) in this section. This rule notation is oriented at the model elements of the business model notation. In section IV we introduce additional means to model abstract rules which are the base for the hierarchical arrangement of the rules. Additionally, we present a small part of a rule hierarchy in detail. The complete rule hierarchy for our example process and how to apply this hierarchy for testing is presented in section V.

II. BACKGROUND AND RELATED WORK

The base of an automated checking of process models is the formal description of the semantics of the model type. A formal description of the semantics of both model types BPMN [5] as well as EPCs [6] is available.

Different projects provide concepts to apply model checking techniques for the verification of the formalized process models. Some approaches transform the process models into the intermediate representation Petri nets, which are then checked. Applying this approach to EPC models is presented here [7]. Similarly, this approach has been applied to BPMN models.

An alternative is to use a textually defined state machine, which is not limited to the semantics of a specific model like the Petri Nets. In our work we prefer this concept since we use the state machine to define additional constructs as presented in this paper. The tool we use is BAM [8], [9]. The BAM concept relies on previous work of the direct transformation of business processes models to verification models for model checkers (SMV Kripke structures) [10].

Error detection techniques based on patterns in processes are presented in [11]. Here, specific process patterns are considered. These patterns may be expressed as Prolog rules [12]. This makes clear that not only the temporal sequence of system is issue of quality assurance but also the static dependencies have to be considered.

The rule may be expressed in the language provided by the checking tool. However, domain experts who are aware of the processes and rules may not be aware of these languages. A graphical rule model would support the domain experts. [13] points out the importance of a graphical representation of the
models. Further approaches providing graphical rules which allow persons not being experts in temporal logic to model these are [14], [15] and [16]. The latter is the base of our work.

An interesting observation is that in the development of business systems often process models (or parts of the models) are reused or reconfigured into new systems [17], [18]. This means that we have to consider different variants of systems and need to support them and the configuration of systems from existing reference processes. This leads to the idea that rules may also be configurable.

III. BUSINESS APPLICATION MODELLER (BAM)

The BAM version for this paper provides two integrated editors – one to model EPCs and the other for the rules (based on EPC). The validation mechanism is hidden from the user in the background. If an error occurs in a validation run of a rule this error is presented in the EPC model as well.

A. ARIS EPC Notation

An overview on the EPC notation may be found in [2] or [19]. In the paper we present a limited set of EPC elements, which are sufficient for this paper.

Figure 1 depicts the sub-process we use as example in this paper. This EPC model (and the G-CTL rule models) in this paper are created with BAM. The presented process is a simplified purchase process. Before this sub-process starts the customer has already chosen the product(s) she or he wants to buy. The items in the shopping cart represent these products. The element shopping cart is a EPC data element (data storage represented by a red rectangle). After the customer has entered the customer data, he selects the payment method, an invoice is created, the product(s) are shipped and finally the customer data is to be deleted since it may not be used further. This is a privacy requirement of the customer.

The elements used in the diagram are the already mentioned data elements and a sequence of functions (rounded green rectangles) and events (magenta hexagons). These are connected by the control flow (straight black arrows) as well as branches and joins (which are in our case XOR only).

Dashed arrows are connecting the data storage elements with functions (rounded green rectangles). An arrow directed to the function means that the function consume data. Is the arrow directed to the data storage data is stored.

B. Graphical Rule Notation

The graphical rule notation is based on a variant of the temporal logic—the branching-time logic Computation Tree Logic (CTL). In order to support the graphical modelling of the rules we use G-CTL (Graphical-CTL), cf. figure 2 [20]. Similarly to CTL, G-CTL provides temporal connectives as a pair of symbols which are one path quantifier and one temporal operator. The path quantifier determines if an property must hold on all path of a tree (All-quantifier) or must

![Fig. 2. Graphical operator symbols of G-CTL.](image-url)
hold only at least for one path in the tree (Exist-quantifier). The operators specify if a statement has to be true in the next (X) state or eventually (F) in a future state, globally (G) in every state of the path or that a statement has to be true in a path until another statement becomes true (U).

In each case one path quantifier is combined with exactly one operator. This combination determines a basic CTL rule. Examples are AG (each and everywhere), AF (in each path at least once), EG (at least in one path in every state). Only the combination of one path quantifier and one operator is represented by a specific G-CTL symbol.

The Boolean operators and constants (i.e. true, false) can be used in conjunction with these temporal operators. The operands a and b are placeholders for concrete (process) elements. This enables to specify a process in the well-known visual manner and the rules in the very same way.

IV. RULE ABSTRACTION AND SPECIFICATION

An example of such a concrete G-CTL rule is depicted in figure 3. This rule states that always globally (AG) when customer enters data (stored in the customer data storage) it is implied that at all paths (always in the future, AF) there is a delete customer data (from the customer data storage).

This rule is explicitly defined and fits only to a specific process model or a very limited set of models. We consider means to create abstract rules on base of concrete rules.

A. Mechanisms for Rule Abstraction and Element Specification

Temporal logic itself provides already mechanisms supporting abstraction of the sequences in paths. For instance, the path quantifier E (exist) and the linear time operator F (future) do not explicitly define a concrete position in a sequence. Although the concrete position is kept abstract these CTL operators address an unspecific position in one branch one or more steps ahead from a current position.

- **Wildcard**: A further abstraction is the wildcard (*). The wildcard still specifies the type of element. This is in detail either the element type function or event.
  - **Attributes**: Attributes are attached to elements in general and elements with wildcards in particular. Since the wildcards as introduced are rather unspecific further specifications are of use. The number of potential elements is limited to a well-defined set of elements.
  - **Identity**: A set of elements may be determined by wildcards. In G-CTL-rules similar elements are required at different places. For example, working with an unspecific data implies an action in the future when the same unspecific data appears again. The identity marker allows to mark the unspecific data and then to identify the same data when occurring in the future.

An identity is only valid within the same rule.

Figure 4 presents an example of the usage of wildcards, attributes and identity. Actually, figure 4 depicts no complete rule but only a fragment of a rule (We call this fragment pattern which is symbolized by the rounded rectangle around the function and data element.). The figure shows the abstract representation of the lower delete pattern of the rule in figure 3.

This simple abstract delete pattern consists of a function specified with the attribute delete, which will delete data with the identity i=1 and the attribute pd (private data). The identity i=1 revers to another data element in the rule (probably in another pattern) and is not depicted in the figure. It indicates that these both data elements must be the same what ever they really are.

Compared to the concrete patterns of rule in figure 3 it comes clear that now it is undefined, which concrete function will be addressed as long as this function has the attribute delete. The data element with identity i=1 addresses only elements with the same identity within the same rule.
of the rules in such a hierarchy represents a specific sub-
requirement. In order to visualise the dependencies between
these rules we use the FODA notation (Feature-Oriented
Domain Analysis) [21]. Figure 5 shows an example of a rule
hierarchy (actually a segment only) and the notation of the
hierarchy. The rules are arranged at different levels. The top
rule (1.4 delete data previously entered) is rather general.
The leaves are quite concrete. The dependencies between the
rules in the figure are alternative which is an exclusive choice
(logical XOR) and optional (logical OR). Not presented in the
figure is the mandatory relation (logical AND).

Mandatory relations which require that both rules need to be
kept are not part of this detailed section of the rules hierarchy.
The complete diagram with all rules concerning our example
process (c.f. figure 1) is shown in figure 6. This diagram uses
mandatory relations which are symbolised by a simple line
linking the depended rules.

In our example the two cross-tree constraint types (requires
and mutex) are not considered. In larger systems such con-
straints may be that pay on delivery requires real delivery (in
our example this is an invariant) and pay on delivery excludes
(mutex) the electronic download (e.g. of software). The logic
operator AND represents requires and XOR represents mutex.

V. APPLYING THE RULE HIERARCHY

An alternative to the hierarchy of depended rules may be
to build one large rule which comprises every rule. This bears
the problem of all monolithic systems: They are quite large
and very hard to handle for human beings. This hierarchy
of rules defines a rule framework as base of a correct rule
configuration. The rule set may be applied (reuse of rules) to
several similar systems and helps therefore in the development
of these systems. The rule framework has to be configured to
the needs of the targeted system and may then help in the
checking of this new system.

BAM supports the checking of models of a system. All
rules pertaining to the system may be defined in one rule set.
Potentially, all rules of this set may be checked at once. As
this would correspond to a rule hierarchy with only mandatory
relations such a constellation would be rather unusual (but
supported by BAM). Rather typical is a configuration where
some rules are selected and others not. The relation types
alternative and optional reflect this.

The user may select a valid configuration of rules. BAM
automatically validates the given business process model(s)
against the rule configuration. If rules are violated the result
is presented in the model editor (further details of the checking
procedure of BAM may be found in [8] and [9]).

The hierarchical arrangement of the rules supports further
opportunities in testing: the top down testing. The tester may start with the rule at the top of the hierarchy. In the further tests the lower rules are activated (and validated) step-wise (each step further rules are checked) until the goal of the testing is achieved. The other way round it is also possible to start the testing from the leaves of the hierarchy. Here, the tester starts with the lowest rules in the hierarchy.

VI. CONCLUSIONS

The tool-based validation as supported by the Business Application Modeller (BAM) enables the checking of a large number of rules. These rules need to be related to each other. We propose a hierarchical order of rules with the relations mandatory, alternative and optional (as well as the cross-tree constraints requires and mutex). Specific elements (wildcards, attributes and identity) are abstraction mechanisms which support the arrangement rules in hierarchies.

By activating (and validating) the rules stepwise a top-down (starting from the top of the hierarchy) or bottom-up, respectively, testing strategy is supported. Additionally, the rule hierarchies may serve as reusable frameworks for checking a number of similar business process driven systems.

REFERENCES


