Verification of Change in a Fragmented Event-Based Process Coordination Environment
Pieter Hens, Monique Snoeck, Manu De Backer and Geert Poels

Abstract—To allow the distribution of control and visibility of cross-organizational process models and to increase availability and performance of the processes, a process model can be fragmented into logically different parts and distributed in the enterprise architecture. Fragmentation algorithms and execution environments which connect the fragmented process model parts together, recreating the original process execution semantics, have been proposed in earlier works. However, a critical challenge that is left open is the ability to independently change the control structure of a process fragment. This is not trivial as changing the control structure of a specific fragment could break the global process execution. The global process overview is also not available anymore because it is fragmented in the enactment environment and many independent control structure changes could have already been done. In this paper we describe an approach based on state reconstruction which enables to check the admissibility of changes made to the control structure of a process fragment in a fragmented event-based process enactment environment.

Index Terms—Process Enactment/Execution, Distributed Computing, Process Evolution, Change Management, Model Checking

1 INTRODUCTION

Business process model fragmentation is the process of splitting a process model that was modeled as a whole into logically different, smaller model fragments with the intention to distribute the fragments over different execution and controlling partners [4], [6], [11], [18], [19]. When a process model needs to be deployed into the runtime architecture, the process is first fragmented into different parts, according to a specific fragmentation criterion. Each part can hereafter be deployed onto a dedicated execution entity (e.g. a process engine), hereby distributing the coordination of the process into the IT and enterprise architecture (see Figure 1). There are several reasons for process model fragmentation: distribution of ownership and/or coordination across process model fragments [11], [18]; elimination of a single point of failure during process model execution [6], [19]; and increasing availability and performance of the process model execution [4], [6], [19].

Several architectures have been proposed that each describe a specific process fragmentation technique and distributed runtime environment [4], [6], [11], [18], [19]. In this paper we are interested in the fragmentation techniques that use an event-based decomposition method [3], [5], [12], [17], [19]. These techniques decompose a given process model into different process fragments (parts), where each fragment is complemented with an event rule. The event rule states in which state the execution environment should be before the process fragment can start its execution. The event rules therefore represent the control dependency between the different process fragments. Figure 1 shows an example of a distributed event-based coordination environment. Section 2 explains the event-based process fragmentation in more detail.

Most techniques that use the event-based fragmentation approach use the event architecture to create a loosely coupled execution environment [12], [17], [19], where each process fragment becomes autonomous. Indeed, the trigger of the process fragment (the event rule) is included in the process fragment itself. Because of the loose coupling created by the event-based fragmentation, changes can be made to the event rule of a specific process fragment, without having to stop and restart the entire process enactment environment [12], [21]. Distribution of control and/or coordination across process model fragments can therefore easily be achieved.
The above reasoning is used by many of the event-based process fragmentation techniques. A problem that is however left open is that not all event rule changes can be deployed into the runtime process coordination environment. A change made to the event rule of a specific distributed process fragment could break the entire global process execution. Indeed, the event rules in the distributed coordination environment correspond to process coordination elements of the original process execution (i.e. the control dependencies, e.g. a sequence between two activities). Changing something to a process model element, without taking the consequence of the global process execution into account, could break this global process execution. For example, changing an XOR-join to an AND-join without changing the preceding XOR-split in a process model will create a deadlock in the process execution. Changing an event-rule can therefore not be done arbitrarily, because the correctness of a control dependency change of a fragment depends on other fragments in the global enactment environment.

Therefore, an event rule change for a specific process fragment in the fragmented coordination environment has to be checked for its correctness, before it can be deployed in the runtime environment. As mentioned by the event-based fragmentation techniques, changes can still be made without restarting the runtime environment because of the loose coupling between fragments. But first an additional check has to be implemented to see if a specific change is allowed. This is a non-trivial problem as the original process model is not available anymore. The process model is scattered (fragmented) in the distributed enactment environment. Moreover, it is possible that many event rule changes have already been done for other fragments, making the correctness checks even more complex.

This paper proposes a technique to check the correctness of an event rule change in an event-based fragmented process coordination environment. It is based on state space checking techniques for process model analysis. The technique allows to check any event rule change, where event rules can be expressed in first order logic [19] as well as in linear temporal logic [17], [24]. In the next section we provide some background information about the distributed event-based process enactment environment. Section 3 discusses the autonomous change features in this distributed environment. Section 4 describes how the checks can be performed in the distributed enactment environment by providing an algorithm for state space reconstruction. Section 5 explains our prototype and a protocol to concertize the change features in the distributed environment. Section 6 evaluates our approach by formally proving the correctness of the approach, explaining instance specific changes, investigating the allowed local changes, experimenting with the performance and discussing the assumptions made. In the last section we provide the reader with some concluding remarks and future research.

2 Background: Distributed Event-Based Process Coordination

In this section we give a brief overview of distributed event-based process execution and describe the execution architecture required to achieve the distributed execution. Note that many other fragmentation and distribution techniques exist (see the previous section). We focus on the event-based fragmentation as these techniques allow the strict distribution of control and responsibility through the inclusion of the control dependency (the event rule) in the fragment itself.

In a distributed coordination approach not one, but multiple process engines are used to manage, in collaboration, the entire process flow (see Figure 1). At deployment time a transformation algorithm takes as input the process flow as modeled by the process modeler and outputs different process model fragments. These fragments are deployed on different, dedicated process engines (see Figure 1). The global process execution for a certain process instance is performed by the collaboration of all the process engines, each running a different process model fragment.

To achieve this collaboration an event-based communication paradigm can be used [3], [5], [12], [17], [19]. In an event-based communication architecture, components communicate with each other by generating and receiving event notifications, where an event is any occurrence of a happening of interest [21]. Components publish notifications into the architecture, from where they are routed to other interested parties. This routing is done by an event service that keeps track of which entity is interested in which event type and which entity is able to publish which event type (content-based many-to-many routing). In our case, the fragmented process engines are both publishers and subscribers of event notifications, with an event being a past happening in the information system with a business meaning, e.g. the completion
of a business task, the arrival of an order request, the cancellation of a task, etc.

A consequence of using event-based communication is that for each process fragment it must be specified for which event types it needs a subscription (and thus a notification). This specification is called the event rule for the process fragment. The event rule specifies a logical combination of events that describes in which state the (business) environment should be before the process fragment can start executing. The process fragment is enabled if its event rule evaluates to true, i.e., the fragment’s process engine received event notifications indicating the completion of other fragments on which it is sequence dependent, or in other words: the environment is in a state where the fragment is able to execute. The event rules are expressed in first order logic or temporal logic. For example an event rule: A-Completed AND B-Completed states that tasks (or fragments) A and B have to be completed before the rule (trigger) is enabled. We also allow event rules to be expressed in temporal logic (LTL) as this is necessary to allow the fragmentation and expression of more advanced workflow concepts in the event-based environment [17]. Moreover, user defined data conditions can also be included in the event rule. These originate from XOR-splits in the original process description (see [17]). For example, A-Completed AND price>100 is a valid event rule. The latter event rule is only enabled when the A-Completed event is received and the data condition validates. Hereafter we use the following notation to indicate a fragment with its corresponding event rule: FragmentName<Event-Rule AND Cond(user-defined-condition)>

To simplify the presentation, in this paper we specifically look at the most fine-grained fragmentation: task-level fragmentation. Each task defined in the original process model becomes a small fragment on its own, with an included event rule. Note that other fragmentation techniques are also possible, in which process fragments can consist of several tasks [6], [11], [18].

3 LOCAL PROCESS FRAGMENT CHANGE

When a process coordination is fragmented and distributed in the environment, a process manager can be assigned for each specific process fragment. Because the start logic is included in each fragment, process managers can be made responsible for their own process fragments and have the ability to adapt the control dependency of their fragment. This is called a local change for a fragment and is defined as follows:

Definition 1: A local process change for a specific fragment is a change where:
- a change is made to the control dependency of a fragment; and
- the change is made autonomously, i.e. the change is not propagated to other fragments and is made without knowledge of any other running process fragments.

Changing the control dependency of a fragment is supported by changing the event rule of that fragment. Indeed, since the event rule describes in which situation (process state) the fragment is enabled for execution, changing the rule will change the control dependency of the fragment. In some cases, this also means that the position of the fragment in the global control flow will change. For example, if the event rule of a fragment is changed from A-Completed to B-Completed, the position of the fragment in the global control flow changes from after-A to after-B. The event rules can be changed at free will, as long as the alphabet (i.e. existing event notifications) of the current process enactment environment is used. The possible changes to an event rule are discussed in section 6.3. Elements can be deleted, added or switched in the event rule. Moreover, new fragments can also be introduced.

The autonomous change property of the local change defines that the change is performed at the to-be changed fragment itself, without propagation to other fragments. This means that a local change is not visible to other fragments and a change is only made in the to-be redeployed fragment. A local change can however not be done arbitrarily, as the correctness of the change is dependent on the other process model fragments in the enactment environment. Repositioning a fragment in the global process’s control flow could therefore affect the correct execution of the global process flow, resulting in deadlocks or never enabled activities. For example changing a rule A-Completed OR B-Completed to A-Completed AND B-Completed could produce a deadlock when the execution of tasks A and B is mutually exclusive.

Care therefore has to be taken when changing the local event rules for a certain fragment. Modification of a fragment’s control dependency should not interfere with the correct execution of the global process flow. On the other hand, as much modifications as possible should be supported by the local change operations. A correctness criterion therefore has to be employed, describing when a local change can be performed with respect to the execution semantics of the business process. Note that besides the check of correct execution semantics, other checks also need to be implemented. The change should not violate business constraints/rules, access regulations should be enforced (not everyone can change the local rule), etc. The focus of this research is however on the first topic: checking the correct execution semantics of the global process after local change. Also note that the described local change is only useful when business fragmentation is performed [18], i.e. the division in fragments has a business meaning. If fragmentation is
4 Correctness of Local Event Rule Modifications

As noted in the previous section, a deployment criterion is needed which checks if a local event rule modification is allowed according to the global process execution. If the modification is accepted by the correctness criterion then the modification can be deployed into the runtime architecture without any problems. In this paper we focus on the first two properties of soundness as a correctness criterion. This is also known as weak soundness [20].

We define the correctness criterion for a local change as follows:

**Definition 3:** A modification of an event rule for a specific process model fragment is allowed if and only if the resulting global process model is weakly sound.

Checking the soundness criterion cannot be done locally with only the new to-be deployed event rule as information. An external entity needs to be employed which can reconstruct the global process model and check if soundness is valid. The reconstruction is needed because multiple local changes could have already been deployed over the course of the lifetime of the process type, therefore making the originally deployed global process model obsolete. The reconstruction needs to be done with all event rules of all fragments involved in the process type’s execution. This reconstruction is however not trivial as new event rules can be arbitrarily constructed. The resulting reconstructed global process can therefore contain not-bounded places, infinite loops, non-mutually exclusive XOR data conditions, etc. This degree of freedom makes the reconstruction of the global process model and therefore the soundness checking an intricate task.

Checking the soundness conditions on a process model is usually performed by means of a reachability analysis. For this reason, the reachability graph for the process model needs to be constructed. The reachability graph represents the state-transitions of a process model, where the nodes are states of the process model execution and transitions represent the execution of a chosen activity in the process model. Because an event rule represents a state of the process execution environment, they map directly onto a state graph. Instead of first constructing the global process model out of the existing and new event rules and hereafter calculating the reachability graph of the constructed process model, the reachability graph can be directly constructed using the event rules.

A process modeler has the responsibility to design sound models. This will guarantee that instances of the model execute in a safe and correct way. In the same way the soundness criterion can be used to check the correctness of a change made by the local fragment manager. If the reconstructed global process model is sound, the change can be adopted. After modification deployment there are no deadlocks, live-locks and no dead transitions, i.e. every activity in the originally defined process model is executed in at least one process instance.

The non-existence of dead activities is however a too strict assumption that has no need in the distributed execution environment. Creating a dead activity by performing a local change can be desired behavior. It is possible that a local change manager wants to eliminate the execution of another fragment in the global process execution. When a dead activity is created by a local change, a warning can be issued to the local change manager, but the existence of the dead activity should not restrain the change deployment. We therefore focus on the first two properties of soundness as a correctness criterion. This is also known as weak soundness [20].
all fragments and their event rules, the state graph of the process execution environment can be constructed and used to evaluate the soundness criterion. Note that this is only possible if the generated event rules are correct and the original process modeler follows the rule of creating mutually exclusive data conditions that originate from the same XOR-split (see Section 4.2.1). For the rest of this research we therefore assume a sound process execution environment as starting point. A local change is issued in this (correct) execution environment.

The following section describes how to construct the reachability graph and explains how the resulting state graph can be used to check the soundness criterion.

4.2 Construction of the Reachability Graph

The reachability graph is represented as a finite state machine, having as input alphabet the fragments that can be executed in the enactment environment. A state in the graph represents a state of the execution environment where certain fragments have already been executed and where other fragments are enabled for execution. Note that the enablement of a fragment depends on the event notifications which have been published in the environment and on the validation of any included data condition (see Section 2). Before the algorithm for the graph construction is presented, we first describe how these data conditions are handled in the graph construction. Next, the properties of the reachability graph are discussed. In the final section, the algorithm is presented.

4.2.1 Data Handling in the Reachability Graph

As mentioned before, a fragment is only enabled when its event rule validates and the included data conditions validate. For example, for the following event rule: \( B < A\text{-Completed AND } \text{Cond}(x>5) > \), fragment \( B \) is only enabled when an \( A\text{-Completed} \) event is received and when the data condition \( x>5 \) validates. When building the reachability graph, the possible values of \( x \) in the previous example are not known as \( x \) is only instantiated at runtime. To still be able to incorporate the data conditions in the graph construction and state inspection, we make the following assumptions: every time a data condition needs to be evaluated, it either validates or not; the data condition is evaluated each time an XOR-Split is reached, no matter any prior validation of the same condition; and data conditions originating from the same XOR-Split are mutually exclusive. The first two assumptions create an exhaustive state graph. The last assumption is compatible with the description of the Exclusive Choice workflow pattern [33]. These assumptions are discussed in more detail below:

1) Validation of a data condition: Since the possible values of a data condition are not known at design time, every time a data condition is evaluated, we assume that it either validates or not. This means that if the execution environment reaches a state where a data condition is evaluated it has two possible succeeding states, one where the condition is true and one where the condition is false. However, to limit the number of states in the graph (see also the performance evaluation in Section 6.4), the false conditions are not explicitly added to the reachability graph. Section 4.2.4 describes how the impact of a false condition can still be investigated on the resulting graph.

2) Reset of the data condition each time the XOR-Split is reached: When a condition is validated, it stays validated (true) in all succeeding states until the condition needs to be checked again. For example in Figure 2, the condition \( x>5 \) validates when fragment \( B \) is chosen for execution and remains validated until fragment \( A \) is executed again (\( x \) can again be greater or smaller than 5). We therefore assume that every time an XOR-split gateway is triggered, the data condition is evaluated (again); and according to the previous assumption, the data condition can either (again) validate or not. This assumption creates an exhaustive state representation with respect to loops in the process model. In the construction of the reachability graph a mechanism therefore needs to be added that can reset conditions whenever necessary (see Section 4.2.3).

3) Mutually exclusive data conditions: It is assumed that data conditions originating from the same XOR-Split are mutually exclusive and complete [33]². For example, in Figure 2 either \( B \) or \( C \) can execute after \( A \), but not both. Since the mutual exclusivity cannot be checked automatically, we need a way to relate a data condition to a specific XOR-Split. This information is however lost in the event-based fragment enactment environment.

For example, in Figure 3, every fragment gets translated to a similar event rule, albeit a different data condition. In the runtime environment this poses no problems, because the data condition is checked each time the event rule is triggered (an X-Completed event is received). The start of the fragment will therefore only be triggered when the event is received and the data condition is validated. This (data-)check is

Fig. 2. Example of a data condition that is reset
However, not possible when constructing the reachability graph, because the value of the variable in the condition is not known. If we assume that all these fragments are mutually exclusive because they have the same event rule (X-Completed), but a different data condition, then the reachability graph would not be correct. The process model in Figure 3 allows the execution trace XAC, whereas our previous assumption would only allow the traces: XA, XB, XC or XD. A different approach is therefore needed which relates the data condition included in the event rule of A to the data condition included in the event rule of B (A and B are mutually exclusive and can execute in parallel with C and D).

For this purpose, the originator of a conditional check (the XOR-Gateway) should be included in the data condition in the event rule. The transformations proposed in existing research on process fragmentation therefore have to be extended [12], [17], [19]. They have to allow the inclusion of the unique ID of the gateway that gave rise to the addition of a data condition in the generated event rule. The definition of a data condition in an event rule is therefore extended as follows:

**Definition 4:** A data condition in an event rule is a tuple: (Cond,Orig) where:
- Cond is the actual user defined data condition.
- For example: x>5
- Orig is the (unique) id of the XOR-split gateway that gave rise to the addition of Cond to the event rule.

For the example depicted in Figure 3, the event rules of the fragments would be changed as follows (assuming XOR1 and XOR2 are the IDs of the XOR-gateways in the process model):

- A<X AND Cond(cond1,XOR1)>
- B<X AND Cond(cond2,XOR1)>
- C<X AND Cond(cond3,XOR2)>
- D<X AND Cond(cond4,XOR2)>

During the construction of the reachability graph, it is now assumed that A and B are mutually exclusive because they have different data conditions that originate from the same XOR-Split. On the other hand, A and C are not tagged as mutually exclusive because their (different) conditions originate from different XOR-Splits.

Note that when a local change is issued that adds a data condition to an existing event rule (or creates a new rule with a data condition), the originator-ID of the newly added data condition is assumed to be different from all other data conditions in the enactment environment. A newly added data condition therefore always originates from a unique XOR-Split. This is due to the fact that a local change is performed without knowledge of other fragments and a new data condition can therefore not be linked to a specific gateway (for a discussion see Section 6.5).

### 4.2.2 Properties of the Reachability Graph

To capture the event-based executional state information (which fragments are enabled for execution in a certain state), a state in the reachability graph contains two elements:

1) All event rules, linked to their respective fragment are kept in each state of the reachability graph. This is done to represent the state of the event rules at a certain state in the reachability graph. Figure 4 shows an example of the possible states for an event rule \( A \land B \) for a fragment \( F \). An event at a specific state of the event rule can be disabled (0), enabled (1) or unbounded (\( \omega \)). An event becomes unbounded from the moment that it is enabled more than once (a similar reasoning is used in building coverability graphs [34]). The boxed states in Figure 4 are accepting states for the event rule. An accepting state is a state where the rule validates, hereby enabling the execution of the corresponding fragment. When the corresponding fragment executes, all enabled events are cleared (consumed). A similar state construction can be done for LTL rules using a Rabin automaton [2], [14].

2) The user defined (data) conditions which are valid at this state are stored in the state. As noted earlier, the condition stays valid in any succeeding states until the condition is evaluated again. To know when a data condition is evaluated again, the event
rule that triggered the validation of the condition is also stored in the state. For the example in Figure 2, when fragment B is chosen for execution, the data condition \(x > 5\) is stored in the current (and succeeding) execution state. Moreover, the event rule that gave rise to this validation: A-Completed is linked to the validated condition in the state. If a new execution state is reached where the corresponding event rule validates (i.e. fragment A is executed), the contained condition is reset (removed). Hereafter fragment B can be executed again or the other path in the process flow can be chosen \((x \leq 5)\).

States are linked together in the graph by a transition, containing one element of the alphabet. The transition represents a fragment that is chosen for execution and which changes the state of the execution environment.

Formally, the reachability graph and its states are defined as follows:

**Definition 5:** The reachability graph in the distributed event-based execution environment is a tuple \((\Sigma, S, s_0, \delta, F)\):

- \(\Sigma\) is the input alphabet, which consists of all (completion-, start- and cancel-) events of fragments of the process type;
- \(S\) is a finite set of states;
- \(\delta : S \times \Sigma \rightarrow S\) is the state transition function;
- \(s_0\) is the initial state (which equals the trigger for the start of the process);
- \(F \subseteq S\) is the set of final states accepted by the process model.

**Definition 6:** A state \(S\) from the reachability graph is a tuple \((\mathcal{ER}, C)\):

- \(\mathcal{ER} = \{(\text{fragment}, \mathcal{ER})\}\) the set of fragments with their corresponding event rule and rule-state;
- \(C = \{\text{(ER, (Cond, Orig))}\}\) the set of data conditions validated at this state, linked to the event rule that gave rise to the addition of this validated data condition.

As the reachability graph can become infinitely large when loops are involved in the process model, we define the equality of two states as follows:

**Definition 7:** Two states are equal in the reachability graph if and only if all possible further execution paths are equal for both states.

**Theorem 1:** Two states \(s_1\) and \(s_2\) are equal if and only if:

- the state of every event rule on \(s_1\) equals the state of the event rules on \(s_2\), with:
  - The state of a basic event rule is represented by the enabled events in the event rule (see Figure 4);
  - The state of an LTL rule is represented by its state in its respective Rabin automaton [2], [14], [25]².
- the validated data conditions present on \(s_1\) equal those present on \(s_2\).

**Proof:** We prove the first bullet of the theorem: assume we have two states \(s_1\) and \(s_2\) that are different, but nevertheless have equal event rule states. Since the states are different, according to definition 7, there exists an execution sequence \(\sigma_{1..i}\) where \(\sigma_i\) is the point at which the execution parts differ. So, \(\sigma_{1..i-1}\) brings \(s_1\) to a state \(s_k\) and \(s_2\) to a state \(s_l\). \(s_k\) allows the further execution of \(\sigma_i\), but \(s_l\) disallows the execution of \(\sigma_i\). This means that \(\text{state}(\mathcal{ER}_{\sigma_i})\) in \(s_k\) \(\neq \text{state}(\mathcal{ER}_{\sigma_i})\) in \(s_l\), with \(\mathcal{ER}_{\sigma_i}\) the event rule of the fragment \(\sigma_i\). Because an equal execution sequence \(\sigma_{1..i-1}\) is followed to reach \(s_k\) and \(s_l\), and the state transitions within a single event rule are deterministic, the only way to reach a different state for \(\mathcal{ER}_{\sigma_i}\) in \(s_k\) and \(s_l\) is when \(\text{state}(\mathcal{ER}_{\sigma_i})\) in \(s_k\) \(\neq \text{state}(\mathcal{ER}_{\sigma_i})\) in \(s_l\). This is in contradiction with our initial assumption that \(s_1\) and \(s_2\) have the same event rule states. We thus can conclude that the assumption that \(s_1\) and \(s_2\) can have execution paths that differ at some point is wrong. So, when their event rules states are equal, the execution paths from these states will be equal and therefore, according to definition 7, \(s_1\) and \(s_2\) are equal. A similar proof can be made for the reverse case (assuming two equal states \(s_1\) and \(s_2\), but having different event rule states).

### 4.2.3 Algorithm for the Reachability Graph Construction

Algorithm 1 shows the algorithm to construct the reachability graph. The algorithm takes as input a set of fragments linked to their corresponding event rules and builds the reachability graph in a breadth-first manner. Initially, a start state is created, which is included in an empty graph (lines 1 and 2). The start state contains one event in its event trace: the start trigger of the process³. The states of all event rules are updated to correspond with the initial situation. For each unvisited state in the current graph, the enabled fragments (those that have an event rule in the accepting state) are retrieved (line 9).

Besides the validation of the control events in the event rule, the data conditions present in the event rule should also validate on the current state. If a data condition is present in the event rule, that condition validates if (a) no conditions are present on the current state or (b) a set of validated conditions is present on the state which include the conditions in the event rule and which originate from the same XOR-Split. As noted before, because the alphabet of a data variable is not known, we assume that two conditions, originating from the same XOR-split are always mutually

2. The Rabin automaton represents a deterministic state machine for the LTL formula.
3. It is assumed that the start trigger is known by the external manager.
exclusive}. This means that we can compare conditions textually, without knowledge of the possible values of a variable in a data condition.

When all valid fragments are collected, a new state is created for each of these valid fragments (line 13). The new state contains all updated event rules of the previous state (see Figure 4). Any validated data conditions are also added to the new state. As explained above and in Definition 6, we assume that a condition is evaluated each time its corresponding event rule validates. Any relevant conditions are therefore reset (removed) in the new state when their corresponding event rule validates in the new state (lines 18-22).

Before the new state is added to the graph, it is checked if a similar state does not already exists (line 25, see also Theorem 1). Either the new state is added to the graph or a transition is created to an already existing state (lines 28-34).

The previous process is repeated for each validated rule and for each non-visited state in the graph, until no new states are added. Eventually, any not included fragment executions are added as dead (not connected) states in the graph (lines 44-47). These are fragments which are never enabled and therefore never execute.

### 4.2.4 Checks

Once the reachability graph is constructed, the soundness criterion can be checked on the graph and a report can be generated for the local fragment manager. To check the properties, we first need to define the end states in the graph.

**Definition 8**: A state \( s \) in the reachability graph \( \mathcal{G} \) is an end state if and only if every event rule in \( s \) resides in an initial state.

The initial state of an event rule represents a state where all events are disabled, i.e., there is no event notification present in the execution environment which may or may not trigger the execution of a fragment. This can be compared with a token in a process execution environment. If a token resides somewhere in a place between the start and end place of the process model then the process is not in an end state (a task can still be executed, or a deadlock is present). When all event rules reside in an initial state, all event notifications (or tokens) are consumed. This definition corresponds to the soundness criterion. An end state is reached when all possible activities are executed and all execution tokens are consumed. In Figure 4, the left upper state is an initial state for the depicted event rule.

To check the first soundness property, each state can simply be inspected on its possibility to reach an end state, by using the \( \delta \) function of the reachability graph.

The second soundness property can be checked by looking at the leaves of the reachability graph. If there are leaves which contain an event rule that has an \( \omega \) number of enabled events, an unbounded (and unwanted) place exists in the execution environment.

Since we did not include the negative path of a data condition (see the section on data handling), a third check is needed to check for deadlocks that occur because a data condition did not validate. This check can simply be done by examining the end states in the process. If all end states contain the same validated condition, then a deadlock can occur when this condition is not validated. Indeed, every end state of the process execution can only be reached when the

---

### Algorithm 1 Reachability Graph Construction

```plaintext
define \( \pi_X : \{\ldots X, \ldots\} \to X \) as a projection operator which selects an element from a given tuple

**define state** : \( ER \to STATE \): A function that returns the state of a given event rule, with \( \top \) an accepting state

**define update** : \( (\mathcal{E}R, \text{fragment}) \to \mathcal{E}R \): A function that updates the state of all event rules, provided the execution of a specific fragment

**Procedure** constructGraph (input \( \rightarrow \{|\text{fragment,E}\}| \) )

1: \( s_0 \leftarrow \{(\text{StartProcess,update(input,\text{StartProcess}),} \emptyset)\}
2: \( G \leftarrow \{\emptyset, s_0, \emptyset, \emptyset\}\)
3: Leaves \( \leftarrow \{s_0\}\)
4: Visited \( \leftarrow \{\}\)
5: 
6: while Leaves \( \neq \emptyset \) do
7:   for all \( s \in \text{Leaves do} \)
8:     // Find all enabled transitions for the current state
9:     ValidRules \( = \{e | e \in \pi_\text{EF}(s) \land \text{state}(\pi_\text{ER}(e)) = \top \land (\forall \epsilon_c \in \pi_\text{Cond}(\pi_\text{ER}(e)) : \exists s_a \in \pi_\text{C}(s) \land \text{Orig}_c \in \pi_\text{C}(s) \land \text{update}(\pi_\text{ER}(s), \text{fragment}(t)), \pi_\text{C}(s) \cup \{\text{state}(\pi_\text{ER}(e))\})\}
10:     \text{update} \leftarrow (\pi_\text{ER}(t), \pi_\text{Cond}(\pi_\text{ER}(t)))
11:     // Reset the conditions on the newly created state
12:     for all \( c \in \pi_\text{C}(s_\text{new}) \) do
13:       if state(\( \pi_\text{ER}(c) \)) = \( \top \) then
14:         \( \pi_\text{C}(s_\text{new}) \leftarrow \pi_\text{C}(s_\text{new}) \setminus c \)
15:         end if
16:      end for
17:      end if
18:     // Find a similar already existing state in the graph
19:     \( s_\text{new} = \{s_i : s_i \in \pi_\text{C}(G) \land \text{state}(\pi_\text{ER}(\text{new})) = \pi_\text{ER}(s_i) \land \pi_\text{C}(s_\text{new}) = \pi_\text{C}(s_i)\}\)
20:    // Create a transition to the new (or existing) state
21:    if \( \exists s_\epsilon \) then
22:       \( \delta(s, \text{fragment}(t)) = s_\epsilon \)
23:    else
24:       \( s_\epsilon = \pi_\text{ER}(G) \cup \{s_\text{new}\} \)
25:       \( \delta(s, \text{fragment}(t)) = s_\text{new} \)
26:     end if
27:    end if
28:    end for
29:  // Mark the node as visited
30:  Visited = Visited \cup s
31:  end for
32: Leaves = \{s | s \in \pi_\text{C}(G) \land s \notin \text{Visited}\}
33: end while
34: // Add all fragments to the input alphabet (including dead fragments)
35: \( \pi_\text{ER}(G) = \pi_\text{ER}(G) \cup \pi_\text{fragments(input)} \)
36: Result := \( \mathcal{G} \)
```
condition validates. Formally:

A deadlock occurs when \( c \) does not validate and:

\[
(\bigcap_{s \in ES} \pi_C(s)) \ni c
\]

with \( ES \) the collection of end-states in \( G \).

The checks above can be examined by the external change manager and reported to the local change initiator. Additionally, any dead transitions can also be reported (e.g. as a warning).

### 5 Proof of Concept

For a proof of concept, we implemented the algorithm in java/EMF using BPMN2.0 as the process language\(^4\). The algorithm is used on top of our own event-based fragment enactment environment [17], in which an algorithm is implemented which fragments a given process model and uses a publish/subscribe architecture to route the event notifications, hereby triggering specific process parts.

An external change manager is used to handle a local change request, construct the reachability graph and check the soundness criterion. Note that such a central manager does not interfere with the actual distributed execution. When needed, a distributed management infrastructure can also be used [15]. The change manager notifies the fragment of the possibility to change or not. Additionally, warnings can be issued notifying the local change manager of any lazy or unbounded fragments which get introduced by the change. The local change manager can hereafter still decide whether he wants to continue or not.

### 6 Evaluation

In the next section we formally validate the correctness of the graph construction. The following sections discuss the allowed local process changes and Section 6.4 describes the performance of the reachability graph construction. Finally, the limitations of the data condition assumptions are discussed.

#### 6.1 Proof of Correctness

**Theorem 2:** The constructed reachability graph correctly represents the actual distributed event-based process execution environment to be able to check soundness, i.e. all possible execution traces in the execution environment are also allowed by the reachability graph and if the execution environment is bounded, all the execution paths allowed by the reachability graph are also possible in the process execution environment.

The restriction to boundedness is required because of the abstraction made to represent unbounded events. Because the reachability graph uses an \( \omega \) rather than the exact number (see Figure 4), it will allow more traces compared to the actual execution environment. This will however only happen in the case that the execution environment is unbounded and therefore not sound. If the graph accepts unbounded traces, then it will accept more traces than the execution environment, but this indicates that in any case, the environment is not sound. If all events are bounded, the graph is a perfect representation of the execution environment and enables to check the soundness criteria. The following proof therefore demonstrates the correctness of the reachability graph, provided boundedness.

We prove the correctness by first looking at the data assumptions made to capture the XOR-split gateways in the process execution environment. Next, we look at the graph construction based on the event rules themselves.

**Theorem 3:** The reachability graph builds a correct representation of the state space with respect to the data conditions included in the event rules.

**Proof:** Suppose there is a state \( s \) in the reachability graph \( G \) with a respective state in the actual execution environment \( E \). Suppose that we have a fragment \( f \) with an event rule \( er_f \), which includes a data condition \( c_1 \). Suppose that \( state(er_f) \) in \( s \) is an accepting state. Suppose that \( E \) allows the execution of \( f \) in \( s \), because \( c_1 \) validates. Suppose that \( G \) rejects the execution of \( f \) in \( s \):

\[
G : s_1 \xrightarrow{\sigma} s \xrightarrow{f} s_2, \text{ with } state(er_f) = \top \land c_1 = \top
\]

According to line 9 in the algorithm, the only way the execution of \( f \) can get rejected in this case is that there exists a validated data condition \( c_2 \) with the same originator as \( c_1 \) in \( s \): \( \exists c_2 : c_2 \in \pi_C(s) \land \pi_{Orig}(c_1) = \pi_{Orig}(c_2) \land c_1 \neq c_2 \). According to our initial assumption, \( c_1 \) validates in \( s \), although \( c_2 \) with the same originator also validates in \( s \). If we assume \( G \) to be correct up until \( s \) our assumption of mutually exclusive data conditions is violated (see 4.2.1). The assumption is therefore incorrect and \( G \) also allows the execution of \( f \) in \( s \).

**Proof:** Suppose there is an execution sequence \( \sigma \) possible in the actual execution environment \( E \), but rejected by the constructed reachability graph \( G \). Suppose that we have the following situation:

\[
E : s_1 \xrightarrow{\sigma_1...\sigma_i} s_2 \xrightarrow{\sigma} s_1 \xrightarrow{\sigma_{i+1}} s_o
\]

\[
G : s_1 \xrightarrow{\sigma_1...\sigma_i} s_2 \xrightarrow{\sigma} s_1 \xrightarrow{\sigma_{i+1}} s_2
\]

For \( \sigma_{i+1} \), \( E \) and \( G \) agree on acceptance and result both in a state \( s \). \( \sigma_i \) is the first fragment on which they disagree. Assume that \( er_{\sigma_i} \) is not in an accepting state in \( s \) for \( G \). However, assuming that \( G \) was correct until state \( s \), the execution of \( \sigma_{i+1} \) would lead to the same state of \( er_{\sigma_i} \) both in \( E \) and in \( G \) because the state

---

4. Available at http://code.google.com/p/debo/
machine of $er_{\sigma}$ is deterministic (and correct). This means that in $s_i$, $er_{\sigma}$ should either be in an accepting state for both $E$ and $G$ or in a non-accepting state for both $E$ and $G$. All execution sequences accepted by $E$ are therefore also accepted by $G$.

### 6.2 Process Instance Migration

We need to check what happens with the running process instances if a local change is deployed into the architecture. In most change migration techniques, a correctness criterion is employed that defines when a change can be propagated to a specific instance [27].

For a local change however, no criterion is needed because the change can be correctly propagated to any running process instance. We mean by this that a process instance that eventually executes a changed fragment, will be able to use the new event rule of that fragment. Note that changes to the past are not allowed: a process instance that already executed the fragment for which a new event rule is deployed and which will never execute the fragment again, is therefore left running in the old configuration.

**Theorem 4:** After an allowed local change $c$ for fragment $f$, any running process instance $i$ can adopt the change $c$ and reach a valid end state.

**Proof:** Define $G^o$ the original reachability graph before local change and $G^n$ the reachability graph with the local change included. $G^o$ and $G^n$ are sound. Define a path from state $s_X$ to state $s_Y$ in $G$ as a list of transitions $\in \Sigma$ which can be followed from $s_X$ to $s_Y$. $f$ is the fragment that underwent a local change.

We look at two possible cases of a running process instance $i$:

1) $i$ resides in a state of $G^o$, $s_i$, where all paths from $s_i$ to an end state in $G^o$ do not include $f$.
   - As $f$ is not included in any paths leading from $s_i$ to an end state and a change is only made to the control dependency of $f$, the paths from $s_i$ to an end state are not subjected to any change, i.e. nothing is changed for all concurrent fragments that execute after $s_i$. If $f$ was introduced on a path from $s_i$ to an end state in $G^n$, this path also leads to a correct end state, as $G^n$ is sound.

2) $i$ resides in a state of $G^o$, $s_i$, where there exists a path from $s_i$ to an end state in $G^o$ which includes $f$.
   - As a change is only performed on the control dependency of $f$, nothing changed for any previous states not containing $f$, therefore $s_i$ also exists in $G^n$ and because $G^n$ is sound, there exists a path from $s_i$ to an end state in $G^n$. $i$ can therefore migrate to the new global version.

Any process instance will therefore execute correctly, even after the local process change is deployed.

### 6.3 Allowed Process Changes

Using the reachability graph construction and the checks introduced in the previous sections, we can look at which local changes are possible in the execution environment and which changes provide problems. Table 1 shows a (non-exhaustive) set of possible changes, under the assumption that the original process model before local change is block structured. Every XOR-split is therefore followed by an XOR-join with the same elements, every AND-split is followed by an AND-join, etc. The change operations under investigation are the operations possible on a basic event rule. Note that a basic event rule contains four elements: a single completion event, conjunctions, disjunctions and data conditions. We investigate the following operations on these four elements:

1) **creating** a new fragment where the event rule consists of existing completion events. The new event rule has one of the four forms of a basic event rule (single element, conjunction, disjunction, data condition);

2) **deleting** an element from an event rule from an existing fragment. The four forms of a basic event rule are investigated: a fragment is deleted as a whole (single element is deleted), a completion event is deleted which resided in conjunction or in disjunction with other elements and an existing data condition is deleted from the event rule.

3) **adding** an element to an existing fragment’s event rule. Three different operations are investigated. Existing completion events are added in conjunction or disjunction with the ‘old’ event rule and a data condition is added.

4) **switching** elements in an event rule with other fragments. This is a combination of the deletion and addition operators. Four different switch operations are investigated: a single element in the event rule is switched with another (existing) element, a disjunction is switched to a conjunction and a conjunction is switched to a disjunction and a data condition is switched with another data condition.

For each of the changes described above, a number of test fragments is created and the correctness of the change is inspected by means of the creation of the reachability graph. For each change, fragments are created according to a test model which contains fragmented with all four forms of the basic event rule: single elements, conjunction, disjunction and data conditions. For all changes defined above, all possible configurations are checked according to the given model.

From Table 1 we can conclude that the allowed local changes are limited. The only local change that can be performed without any problems is removing an element from a conjunction and adding a new frag-
Indication of allowed local process changes under the assumption of an originally block structured process model

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>OK?</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>+</td>
<td>$N$ is required to execute before process completion.</td>
</tr>
<tr>
<td>$\emptyset$</td>
<td>$N&lt;\phi\land\text{Cond}(\alpha)$</td>
<td>+</td>
<td>$N$ is required to execute before process completion. Cond($\alpha$) is however always validated in the state before execution. If the condition never validates, $N$ is a dead fragment.</td>
</tr>
<tr>
<td>$\emptyset$</td>
<td>$N&lt;\phi\land\rho$</td>
<td>$\pm$</td>
<td>$N$ is required to execute before process completion when both $\phi$ and $\rho$ are executed at some point (in sequence or in parallel). A deadlock is introduced when $\phi$ and $\rho$ are executed in exclusive order.</td>
</tr>
<tr>
<td>$\emptyset$</td>
<td>$N&lt;\phi\lor\rho$</td>
<td>+</td>
<td>$N$ is unbounded and can be executed maximum 2 times (in the assumption that $\phi$ and $\rho$ both validate, otherwise $N$ is bounded to 1).</td>
</tr>
<tr>
<td>$N&lt;\phi$</td>
<td>$\emptyset$</td>
<td>$\pm$</td>
<td>Introduces dead fragments and a possible deadlock. No problem if $N$ is part of an XOR block.</td>
</tr>
<tr>
<td>$N&lt;\phi\land\rho$</td>
<td>$N&lt;\phi$</td>
<td>+</td>
<td>Possible lazy activity $\rho$</td>
</tr>
<tr>
<td>$N&lt;\phi\land\text{Cond}(\alpha)$</td>
<td>$N&lt;\phi$</td>
<td>-</td>
<td>Not possible according to the soundness criterion, unboundness is introduced</td>
</tr>
<tr>
<td>$N&lt;\phi\lor\rho$</td>
<td>$N&lt;\phi$</td>
<td>-</td>
<td>Introduces a deadlock when $\rho$ is chosen for execution at the XOR-split (under the assumption of a block structured model)</td>
</tr>
<tr>
<td>$N&lt;\phi$</td>
<td>$N&lt;\phi\land\rho$</td>
<td>$\pm$</td>
<td>Introduces a deadlock when $\rho$ lies in the downward flow relative to $\phi$. No deadlock is introduced if $\rho$ lies in the upward flow relative to $\phi$ and on every possible execution path.</td>
</tr>
<tr>
<td>$N&lt;\phi$</td>
<td>$N&lt;\phi\lor\rho$</td>
<td>-</td>
<td>Introduces multiple end fragment executions (unboundness).</td>
</tr>
<tr>
<td>$N&lt;\phi\land\text{Cond}(\alpha)$</td>
<td>$N&lt;\phi\land\text{Cond}(\beta)$</td>
<td>$\pm$</td>
<td>Cond($\alpha$) is validated before it can execute. Note that a deadlock is introduced when Cond($\alpha$) never validates and no other mutually exclusive condition is present on another fragment.</td>
</tr>
<tr>
<td>$N&lt;\phi$</td>
<td>$N&lt;\rho$</td>
<td>$\pm$</td>
<td>Introduces dead fragments and a deadlock when $\rho$ lies in the downward flow relative to $\phi$. Introduces lazy activities, but no deadlock when $\rho$ lies in the upward flow relative to $\phi$. If $N$ is part of an XOR-block and contains a data condition, multiple end executions are possible. This is a limitation of the fact that we cannot compare the mutual exclusivity of two conditions. See the discussion at the end of this section.</td>
</tr>
<tr>
<td>$N&lt;\phi\land\rho$</td>
<td>$N&lt;\phi\lor\rho$</td>
<td>-</td>
<td>Not possible according to the soundness criterion, the end fragment is executed multiple times (unboundness)</td>
</tr>
<tr>
<td>$N&lt;\phi\lor\rho$</td>
<td>$N&lt;\phi\land\rho$</td>
<td>-</td>
<td>Not possible, introduces a deadlock</td>
</tr>
<tr>
<td>$N&lt;\phi\land\text{Cond}(\alpha)$</td>
<td>$N&lt;\phi\land\text{Cond}(\beta)$</td>
<td>$\pm$</td>
<td>Not possible if the condition $\beta$ is not mutually exclusive with the other conditions originating from same XOR-Split as $\alpha$, because this will introduce multiple end fragment executions. If $\beta$ is mutually exclusive the change is possible. See also Section 6.5</td>
</tr>
</tbody>
</table>

$\phi$ and $\rho$ are completion events (or a part of an event rule), $\alpha$ and $\beta$ are data conditions and $\emptyset$ is the empty set (the fragment is removed).

6.4 Performance

One of the biggest issues in state space verification methods is the state explosion problem [30]. The number of states in the reachability graph grows exponentially as the number of event rules and fragments increases. Merging similar states already reduces the number of states, but the construction can still take a significant amount of time when the number of possible execution paths increases. We ran an experiment to evaluate the performance of the reachability graph construction and measured the time needed to complete the algorithm for a specific number of fragments\(^5\).

We measured the time needed in the worst case scenario, i.e. all fragments can be executed in parallel and in the best case scenario, i.e. all fragments are run sequentially. For the worst case scenario, $n$ frag-

---

\(^5\) The experiment is run on an Intel Xeon server pc with 64gb of available memory.
ments create $n!$ possible execution paths in the state graph. This means that, in the worst case scenario, even for only 15 fragments more than 5 days are needed to reconstruct the reachability graph (with over 1.3 trillion possible execution paths). However, in the best case scenario for the same number of fragments, only ±40ms are needed to construct the graph. In general, the reachability graph could be constructed in under one minute for process models containing up to 40 million distinct execution paths. It is therefore still viable to reconstruct the graph for a lot of process models (event rule combinations), as long as the number of parallel fragments at one point in the process flow stays limited (~12 parallel fragments). Note however that the graph needs to be constructed for each local change, which harms fast evolving processes.

For instance specific changes, state explosion can be limited by only constructing the reachability graph for future fragments [26]. Indeed, if all future fragments, including the local change, allow the process instance to complete correctly the instance can migrate and the local change can be deployed. By using only future fragments to calculate the state graph, the number of fragments to be inspected can be reduced significantly. This solution is however only possible for instance specific changes. For a global change, the state graph still has to be constructed in its entirety.

### 6.5 Limitations of the Data Condition’s Assumptions

A first limitation is the assumption that data conditions that originate from the same XOR-Split are always mutually exclusive. This assumption excludes the Multi-Choice workflow pattern [33] from the checks that can be performed (which is however supported in the runtime environment itself). A solution to this limitation would be to transform the multi-choice construct to a combination of AND-Split and XOR-Split gateways or to model the data conditions in such a way that they are mutually exclusive [33].

A second limitation is the fact that a newly added (or changed) data condition is always assumed to be linked to its very own XOR-Split gateway. This needs to be done because the new data condition cannot be automatically checked for its mutual exclusivity with the original conditions. The last check in Table 1 provides an example. The reconstructed process model for the change is shown in Figure 5. Only the human change manager can decide if the change is applicable in the enactment environment (if $\beta$ is mutually exclusive with $\eta$). A report of the possible failures should therefore always be provided and the change manager can still decide if the change has to be implemented or not.

6. Note that there are less states in the graph than possible paths, as similar states are merged together.

### 7 Related Work

In process modeling, the construction of a state graph out of the original process model is a common technique to analyze the process [7]. Moreover, the concept of soundness as correctness criterion is commonly used in process modeling [31]. A sound process model is guaranteed to execute correctly. The aforementioned analysis and state graph building are mostly used during the process modeling and analysis phase of process management. In the distributed environment however, the state graph cannot be constructed out of the original process model as the overview is lost in the enactment environment. Moreover, a custom created event rule could not even correspond to a specific process modeling construct.

In rule-based checking techniques, Nazareth [22] proposes the construction of a petri net out of a rule based system to perform a reachability analysis. No complex rules (conjunctions and disjunctions or LTL based rules) are however supported. The proposed transformation rule from rule based system to petri net remains fairly basic. Other rule based verification techniques are more focused on the pairwise comparison of rules [29], in contrast to verifying the chain of activities that our event rules induce. Declarative process modeling approaches like DECLARE [23] also use rules to represent the process flow and use a state graph construction technique to enact the process. The main difference between the declarative approaches and the event-based fragmentation approaches is that the rules in declarative approaches represent constraints (what cannot happen?), while the rules in the distributed coordination approaches represent triggers (when can a fragment start its execution?) [12], [17], [19]. Moreover, the declarative approach assumes an infinite system, while the start of the distributed approaches is a finite process execution.

In the verification of rules, LTL model checking uses automaton [16] to verify certain constraints. LTL rules are translated to a Büchi automaton so as to support the model checking of a system and property by intersecting the automaton of the system and the automaton of the property (LTL rule). The automaton differs with our approach in that we construct a
Finite State Machine, which resembles the reachability graph of the process model. The Büchi automaton on the other hand accepts an infinite number of events and describes the non-deterministic transition of properties which can become true or false. In the distributed process execution approach, no properties can become false. Once an event is received, it stays received for the rest of time. Moreover, our approach integrates mutually exclusive data conditions into the state graph.

The usage of a finite state machine for workflow verification is also used in message interaction conformance checking [13], [28]. State graphs are used to represent the message flow between different distributed (web) services. Rinderle et al. [28] and Foster et al. [13] construct a state graph for each service, representing the incoming and outgoing message flows. The resulting graphs can be compared to check compatibility. In the same way, multi-agent systems check conformance of the participating agents with a protocol expressed in a finite state automaton [1], [8]. The previous approaches differ with our approach, because we need to check the correctness of the sequence dependencies between the fragments (chain of activities) in stead of the conformance of (internal) message acceptance between two or more entities.

In distributed workflow execution, Reichert et al. [26] also describe a system for change support, where a change is initiated at a local process node. The suggested technique however assumes that a local copy of the complete process schema is available at each distributed node. This eliminates the need to reconstruct the global process model or reachability graph to check the correctness of the resulting sequence dependencies after local change deployment. Fdhila et al. [9], [10] also describe change propagation in the fragmented enactment environment. In [9] it is described how changes made to the central orchestration model can be propagated to the different running fragments. This is different from our approach, as changes originate from the local fragments and not from the central model. In [10], Fdhila et al. present change patterns to propagate local (private) changes to other affected fragments. When a specific partner changes its internal process, the impact on the global choreography is computed and the change effects are propagated to each related partner process. The presented approach is interesting as it allows transitive changes to keep the choreography model consistent. The difference with our approach is that we do not allow the propagation of changes to other process parts (see definition 1), the local change has to be autonomous. This is due to the fact that we focus on the fragment enactment environments that implement the event-based execution model. van der Aalst et al. [32] also propose a correctness criteria for dealing with changes in a distributed environment. They present lifecycle inheritance relations to check whether a changed process is a subclass of the original one. Only in this case is the change allowed. The presented relations are focused on the internal workflow structures of the fragments. In contrast, our approach looks at changes to the (unique) triggering relation between the different fragments (the event rule).

8 Conclusion

This paper described a technique to perform an independent control dependency change for a specific process model fragment in an event-based fragment enactment environment. A control dependency change is enabled by allowing the change of the integrated event rule of the fragment. As some local changes can introduce unwanted and incorrect behavior in the global execution, such a change has to be checked against all running fragments in the environment. To check if a change is possible, a reachability analysis is performed on all fragments with their corresponding event rules, including the changed event rule. An algorithm is described to reconstruct the state graph, given all fragments and event rules in the system. The state graph can hereafter be used to check the correctness properties of the resulting system after (possible) local change deployment.

The main disadvantage of the approach is the computational complexity of the state graph construction. As the number of parallel fragments increases, the time needed to reconstruct the graph increases exponentially and becomes at a certain point unfeasible. A lot of process models are however still supported, and the provided technique can still be used to check the correctness in a limited time period.

The described technique allows the executional correct deployment of new event rules. It provides a correctness check that can be used by the fragmentation techniques for process coordination which use an event-based distribution between the different fragments [3], [5], [12], [17], [19].

Future research involves the investigation of extra checks like the incorporation of business rules validation, access regulations, monitoring etc.

References
