Preserving Validity of Batch-job Nets under Change at Run-time

Chris Apfelbeck, Martin Fritz
Custom Solution Development
Application Services Capgemini
Düsseldorf, Germany
{chris.apfelbeck,martin.fritz}@capgemini.com

Jan Jürjens*†, Johannes Zweihoff†
*Fraunhofer ISST, Dortmund, Germany
†Chair of Software Engineering
Technical University of Dortmund
Dortmund, Germany
{jan.jurjens,johannes.zweihoff}@cs.tu-dortmund.de

***Abstract*—In this paper, we develop an approach to preserve validity of executable batch-job specifications during changes at run-time based on Petri-nets. The approach in particular supports changing batch-job specifications while they are being executed, which makes it particularly important to ensure that the change preserves the critical properties. The approach supports verification of the batch-job specifications that are subject to change against these properties and correction of those batch-job specifications that become invalid by the change. The developed approach was implemented and validated in an industrial application context.**

I. Introduction

Changes made to software systems and architecture at run-time are challenging and critical because errors have an immediate impact on the system behavior [1]. If such an error acts unrecognized in a presently executed system, this can be a significant risk. Therefore it is essential to validate changes to be done before they are carried out to prevent errors from execution. This is particularly challenging if the changes should be carried out while the system is being executed. This raises the question how can we apply changes to a running system without putting it at risk.

We consider this problem in the context of batch processing (BP). BP implements the automated handling of data and tasks without the need of an interacting user. The processing can be done without supervision and independently from working hours in an enterprise. BP was first used in times of centralized computing systems and allowed a high utilization of the requested computing resources [2]. But BP is still of importance because of additional advantages like cost reduction, application efficiency, and process integration [3].

The approach for batch processing and the specification of the batches used in this paper supports in addition to processing of batch-jobs also the representation of batch dependencies. These dependencies are for example concurrency, mutual exclusion or serial sequences of batches. The modeling of such dependencies is formally done by the modeling notation of Petri-nets [4], which is extended and adjusted for modeling batch dependencies, resulting in so-called *batch-nets*. An instance of the batch-net runs on a server and has a state in accordance with the status of the scheduled batches.

In order to avoid having to restart the instance when a change has to be made, one needs to let the change take place at while the batch instance is being executed. However, changes may lead to unintentional side effects (e.g. a deadlock which may stop the whole system). It is therefore important to validate the changed batch-net before it is synchronized with the server instance.

In this paper, we present an approach which supports a validation of the change of the instance at run-time. The validation and its result depends on the current state of the batch-net. In case of a negative result, the approach provides a suggestion for how to repair the invalid batch-net and its current state, which makes sure that a valid batch-net is running on the server even after the change.

The approach presented here has been implemented and validated in an industrial application context in a large consulting and IT-service provider, namely Capgemini. Capgemini uses *QuasarBatch* [5] in several of its software development projects, to model and organize batch jobs. QuasarBatch is part of the Capgemini Standard Platform *TECBASE*. TECBASE consists of a number of reusable components, such as logging, persistence and security, embedded in a standard architecture used in productive environments. QuasarBatch is a proprietary implementation of the concept of batch-processing utilizing the concept of Petri-nets in accordance with *Quasar* [5]. Quasar itself is a programming language independent collection of concepts, ideas, terminology, interfaces and components. In addition to the processing of batch-jobs, *QuasarBatch* also supports the representation of batch dependencies.

The remainder of the paper is structured as follows. In section II we define the batch-net modeling notation based on Petri-nets. We then identify critical properties in the context of batch-processing models in section III. For each of these properties, we show how to check that a given change does not violate the property. In section IV, we present an approach which supports repairing a batch-net, which has become invalid through a change. We then validate our research in section V and embed it in the context of related work in section VI. In section VII we come to a conclusion and discuss possible future research.

II. BATCH-NETS

Petri-nets: We recall the definition of Petri-nets (cf. [4]).

Definition 1. *A Petri-net is a 6-tuple,*
$PN = (P, T, F, K, W, M_0)$ *where:*
$P = \{p_1, \cdots, p_m\}$ *is a finite set of places,*
$T = \{t_1, \cdots, t_n\}$ *is a finite set of transitions,*
$F \subseteq (P \times T) \cup (T \times P)$ *is a set of arcs (flow relation),*
$K : S \to \mathbb{N} \cup \{\infty\}$ *: capacity of the places,*
$W : F \to \mathbb{N}$ *: is a weight function,*
$M_0 : S \to \mathbb{N}_0$ *: is the initial marking,*
$P \cap T = \emptyset$ *and* $P \cup T \neq \emptyset$.
A net structure $N = (P, T, F, K, W)$ *without specific initial marking* M_0 *is denoted by* N*; a Petri-net with a given initial marking by* (N, M_0)*. A marking of the net is denoted by* M*.* $\mathcal{M}(N)$ *denotes the set of all markings on* N.

A transition t of the net N marked with M can fire if and only if the following inequalities hold:

$$\forall s \in \bullet t : M(s) \geq W(s,t),$$
$$\forall s \in t\bullet : M(s) + W(t,s) \leq K(s).$$

Here $\bullet t := \{p | (p,t) \in F\}$ denotes all incoming places of t and analogous $t\bullet := \{p | (t,p) \in F\}$ all outgoing places. $M(s)$ is the number of tokens on the place s with respect to a given marking M. If a transition t is able to fire, we call it *enabled* and denote it by $M[t\rangle$. When a transition fires, tokens are created and consumed in the outgoing and incoming places according to the weight function on the arcs and the capacity of the places. The transformation of the marking from M to M' caused by the firing of transition t is denoted by $M[t\rangle M'$. The new marking M' is defined as follows:

$$M'(s) = \begin{cases} M(s) - W(s,t), & \text{if } s \in \bullet t \backslash t\bullet, \\ M(s) + W(s,t), & \text{if } s \in t\bullet \setminus \bullet t, \\ M(s) - W(s,t) + W(t,s), & \text{if } s \in t\bullet \cap \bullet t, \\ M(s), & \text{else.} \end{cases}$$

If M' is obtained from M by transformation t, then we also use Mt as notation for the new marking M'. Furthermore a resulting marking is defined for sequential firing of transitions. Let $w = (t_i)_{i \in \{1,\cdots,n\}} \subseteq T$ be a finite sequence of transitions, then $M'' \in \mathcal{M}$ is a *resulting marking* of M_0 under w if the following holds:

$$\Big(w = \square \wedge M'' = M_0\Big) \vee$$
$$\Big(\exists M' \in \mathcal{M}(N) : M_0[t_1, \cdots, t_{n-1}\rangle M' \wedge M'[t_n\rangle M''\Big),$$

where $\square$ is the empty transition. In this case w is called *enabled*, $M_0[w\rangle$ for short, and we call w a *firing sequence*. The set of all firing sequences is denoted:

$$Occ(N, M_0) := \{w | M_0[w\rangle\}.$$

The set of all *reachable markings* starting from the initial marking M_0 in the Petri-net N is denoted by

$$[M_0\rangle_N := \{M_0 w | w \in Occ(N, M_0)\}.$$

If it is clear which Petri-net is meant, we also write more briefly $[M_0\rangle$. A transition t of a Petri-net is called *potentially firable*, if there is a resulting marking from M_0 so that t is enabled. If a transition t is not potentially firable, i.e. there is no marking in $[M_0\rangle$ so that t is enabled, we denote t as *dead*.

An important property of a Petri-net for our approach is the existence of a *home state*:

Definition 2. *With home state we denote a marking* $M \in [M_0\rangle$ *which is reachable from any other marking* $M' \in [M_0\rangle$*:*

$$\forall M' \in [M_0\rangle : M \in [M'\rangle.$$

We will also need the concept of a *self-loop*:

Definition 3. *A place* p *and a transition* t *form a self-loop, if* p *is both incoming and outgoing place of* t.

Batch nets: To model batch dependencies with Petri-nets, we need some extensions regarding the scheduling of batch jobs. The idea is to couple the execution of a batch job to the firing of a transition. Additionally, we need to limit the execution of a batch job to specific time points (i.e. the transition may only fire at predefined time points), which is not supported by ordinary Petri-nets. The modification we made to Petri-nets is the introduction of special *batch transitions*, *timer places* and *conditional arcs*. The timer places ensure that we have full control of the scheduling of the batch-net if and only if the timer places do not have incoming arcs. This kind of extended Petri-net is called a *batch-net* in the following. This extension is used within the industrial environment TECBASE for modeling batch-job dependencies using batch-nets. We now give a formal definition as an extension of Petri-nets which reflects the above listed considerations.

Definition 4. *A batch-net is a 7-tuple,*
$B = (P^*, T^*, F^*, C, K, W, M_0)$ *where:*
$P^* = P \cup P_T$ *is the set of all places with* $P = \{p_1, \cdots, p_m\}$ *the set of ordinary places and* $P_T = \{p_1^T, \cdots p_n^T\}$ *the set of timer places mentioned above.*
$T^* = T \cup T_B$ *is the set of all transitions with* $T = \{t_1, \cdots, t_k\}$ *the set of ordinary transitions and* $T_B = \{t_1^B, \cdots, t_l^B\}$ *the set of batch transitions.*
$F^* \subseteq (P^* \times T^*) \cup (T^* \times P)$ *is the set of all arcs.*
$C = F^* \to X$ *: mapping of arcs to a set* $X = \{x_1, \cdots, x_j\}$ *of conditions.*
K, W, M_0 *: compare Def. 1.*
$P^* \cap T^* = \emptyset$ *and* $P^* \cup T^* \neq \emptyset$.

We now explain the semantics of these modifications.

A timer place can generate tokens autonomously (in contrast to an ordinary place). This happens in regular predefined time intervals using the format used to define `cron`-jobs in Unix [6]. This format allows the specification of the minutes, hours, days of a month, months, and working days at which the place should create a token (for which wildcards (*) can also be used). Using this format, one can define time points, sets of time points, and time intervals (see table I for examples). In the graphical representation of the batch-net, the timer place will be annotated with the letter "T" (see figure 1a).

m	h	D	M	W	meaning
*	*	*	*	*	every minute, every day
0	23	*	*	6	every Saturday 11 p.m.
0, 30	12-15	*	*	1-5	every working day, every half our from 12 a.m. to 3:30 p.m.

TABLE I: Example: Definition of time intervals (m= minute, h=hour, D=day of month, M=month, W=weekday.

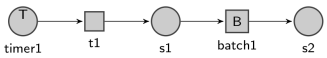
Additionally to the ordinary transitions we introduced special *batch transitions*. The behavior of such transitions when firing is similar to classical transitions. When firing, a batch transition additionally starts a batch job. The tokens in the outgoing places will be created only when the batch job is done. In contrast, classical transitions create and consume tokens without delay. In the graphical representation the batch transition will be annotated with the letter "B" (see figure 1a).

In batch-nets it will be necessary to react on the result of a batch execution. For example the execution of a batch job could lead to an error, or the batch job might take too much time. Therefore we extended the Petri-net concept by *conditional arcs*, represented by the Condition Function C in Definition 3, which maps every arc to a set of conditions. Transitions with outgoing conditional arcs are called *conditional transitions* (see figure 2a for an example).

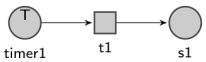
Remark 1. *In this paper we only make use of unbounded capacities in the batch-nets: $K(p^*) = \infty \ \forall p^* \in P^*$. The reason is that the analysis techniques described in section III need this precondition. Murata [4] describes a transformation from Petri-nets with bounded capacity to Petri-nets with unbounded capacity. In this way it is ensured that the precondition is not a restriction to our approach.*

Transformations to ordinary Petri-nets: To check the properties to retain under change, we transform the batch-nets to ordinary Petri-nets in order to check the properties with known algorithms for Petri-nets. As we will see, these transformations are no restriction of generality for our approach.

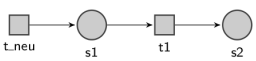
First we determine how to treat the timer places when analyzing the batch-net. Timer places create tokens in regular time intervals. At the initial marking M_0, the timer places are empty (see figure 1b). If **timer1** was an ordinary place, the transition $t1$ would be dead. Because **timer1** is a timer place, after a certain time a token will be created so that $t1$ is able to fire. To simulate in an ordinary Petri-net, that at a given timer place a token can be created at a certain moment, we put a transition without incoming arcs right before the timer place. Such a transition can fire at every moment and thus creates a token on the timer place. In the following definition we formalize this method and denote it by *transformation A* (compare figure 1c).



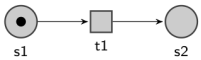
(a) Exemplary timer place and batch transition



(b) Original batch-net



(c) Transformation A



(d) Transformation B

Fig. 1: Batch-nets and transformations to Petri-nets

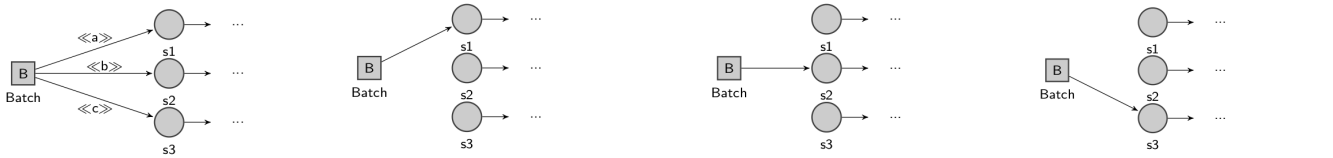
Definition 5. *Transformation A creates a transition t with $t\bullet = s$ for every place s which represents a timer place in the batch-net. Furthermore it defines $M(s) = 0$ for every such s, where M is the current marking of the batch-net.*

An alternative way to transform a timer place is to replace it by an ordinary place and put a token on it. In this way the transition $t1$ is also able to fire at every moment, as desired. This second transformation is necessary since not every property-related analysis leads to a correct result with both transformations. In the following definition we formalize this alternative method and denote it by *transformation B* (compare figure 1d).

Definition 6. *Transformation B substitutes every place s which represents a timer place in the batch-net by an ordinary one. Furthermore it defines $M(s) = 1$ for every such s, where M is the current marking of the batch-net.*

Next, the transformation of batch transitions is very simple, because we merely regard the batch transition as an ordinary one. However, we need to take into account that possibly batches are currently being executed at the transition. As a consequence, the tokens in the incoming place have already been consumed and the tokens in the outgoing place have not been created yet. Therefore, a corresponding number of tokens is created in all outgoing places of active batch transitions. This way, we can analyze batch transitions even at run-time.

Third, we consider conditional arcs, which are not included in ordinary Petri-nets. For every arc of a conditional transition we create a new Petri-net. This net is a clone of the original net, except that only one of the conditional arcs is included in each net (as an ordinary arc) and the other arcs are removed. We call this process *isolating* and the resulting nets *virtual nets*. Figure 2 illustrates the isolation of arcs a, b and c. If in the virtual net several conditional arcs occur, the process must be performed recursively for every virtual net already created. For each additional transition with n outgoing conditional arcs, the number of virtual nets to be created is thus multiplied by n. When analyzing a batch-net (as explained in the next section), every virtual net is checked with respect to the regarded property and subsequently the results are put together to obtain the result for the batch-net. For each property, it will be easily seen, that if the property holds for each virtual net it holds consequently for the batch-net. In this way we can use analysis techniques known for Petri-nets, to analyze batch-nets for critical properties.



(a) Original batch-net (b) Virtual net with cond. arc (c) Virtual net with cond. arc (d) Virtual net with cond. arc

Fig. 2: Conditional arcs and virtual nets

III. Batch Nets: Critical Properties

We now present six critical properties a batch-net has to fulfill initially and under change, to make sure, the batch-job it specifies is executed correctly and consistently, even under change. For each property we explain how to check it. In case of invalidity we apply countermeasures which we present in the next section. The below introduced critical properties have arisen from application context over time and where given by the industrial partner. It has been found that users who are not experts in batch-process modeling, often make mistakes concerning these properties. That is the reason why we picked precisely these ones.

E1: No-incoming-arcs-on-timer-places: The *timerplaces* must not have incoming arcs: An incoming arc means, that other transitions can create tokens in that timer place, which is not desired, since it counteracts the idea of a scheduled batch-net. This property (named *E1*) follows immediately from the definition of the arcs F^* of the batch-nets:

$$F^* \subseteq (P^* \times T^*) \cup (T^* \times P).$$

Nevertheless, after a batch-net is changed, we need to check that this property still holds, which is straightforward since we only have to check every timer place on its own for whether there are incoming arcs or not. For that reason, we do not need any transformation defined in the previous section, when analyzing for this property.

E2: Incoming-arc-for-transitions: In a batch-net, every transition should have at least one incoming place.

$$\forall t^* \in T^* \; \exists \; p^* \in P^* : (p^*, t^*) \in F^*$$

A transition without incoming place can fire at any time, which contradicts the basic idea of a batch-net. We refer to this property as *E2* in the following. Property E2 is easy to check by regarding every transition one after the other and test, whether there is at least one incoming place. As a conclusion, we do not need any transformation to check for property E2.

E3: No-dead-transitions: An important goal in batch-net modeling is to not have any dead transitions in the net, which could lead to a dead-lock of the whole batch-net. The property *E3* denotes the absence of dead transitions:

$$t^* \in T^* \Rightarrow \exists M \in [M_0\rangle : M[t^*\rangle.$$

To analyze this property in a batch-net, we first transform away the timer places using *transformation A*. Thus we ensure that every reachable marking is considered when searching for dead transitions, since the new created transition is able to fire infinitely. To analyze batch-nets with conditional arcs, we construct a virtual net for every conditional arc as explained in section II. For every virtual net, a coverability tree is computed to determine the dead transitions in the virtual nets [4]. A virtual net in which a transition t is not dead exists, if and only if, there is a firing sequence in the batch net, which enables the transition t, assuming that the affiliated condition holds. The other way around, we conclude that a transition which is dead in every virtual net, is dead in the original net (for the given marking). Thus we can determine the dead transitions in batch-nets using approaches for ordinary Petri-nets.

E4: Absence-of-conflicts: A conflict in Petri-nets arises, when several transitions are activated but not every transition is able to fire. After activating one particular transition, the others could be dead, which should be prevented as mentioned above. More generally, batch-nets should model deterministic behavior. Instead, conditional arcs are the appropriate way to split the control flow and to react to several conditions. Thus we aim to exclude conflicts as formalized by property *E4*:

$$\forall p^* \in P^* \text{ with } \left|\{t^* \in T^* | (p^*, t^*) \in F^*\}\right| > 1 :$$
$$p^* \text{ is part of a self-loop with } t^* \; \forall t^* \text{ with } \exists \; (p^*, t^*) \in F^*.$$

To be able to search for conflicts in a batch-net using the corresponding methods for Petri-nets, we transform the timer places to ordinary ones, using either transformation A or B and remove the conditions from the conditional arcs. The absence of conflicts is checked by determining the number of outgoing transitions for every place. If it is greater than one, it is checked whether the transition is in a self-loop with the corresponding place. If that is not the case, a conflict exists. For this analysis technique it is clear, that it doesn't matter which transformation we use since the transformation has no influence on the number of outgoing arcs of an ordinary place.

E5: Bounded: Since we want the batch-net to stop after all batch jobs are done, it should not be possible to accumulate tokens in a place, because this could lead to a non-terminating batch-net. Hence, we require the batch-net to be bounded, as formalized by property *E5*:

$$\forall p^* \in P^* \; \exists \; k \in \mathbb{N} : \left|M(p^*)\right| \leq k \; \forall \; M \in [M_0\rangle.$$

The normal run of a batch-net is as follows: The batch-net is started and works off every single batch job in one run. As a consequence, not more than one token is needed at any place. This is the reason, why we use *transformation B* for the timer places when analyzing for property *E5*. The only exception are timer places, since an unbounded number of tokens can be created there, according to the definition of a timer place

(this corresponds to a queue). But this fact does not influence the analysis process for this property and thus, the choice for transformation B is satisfied. To deal with conditional arcs, the virtual nets are computed and tested one by one for property *E5*. If all virtual nets are bounded, then the original net is bounded too. Assuming, the batch-net B is unbounded. Then, a virtual net exists which is unbounded, because there must be a subnet, in which infinitely many tokens can be accumulated. Vice versa, we see that the original net is bounded, if all virtual nets are bounded. The previous property is closely linked to the next one.

E6: Home states: In every batch-net, there should exist a marking which characterizes that all batch jobs are done and that the batch-net waits for new tokens in the timer places. For this, we can use the concept of *home states* introduced in section II. We need a special home state, where the batch-net will cease activity and waits for new tokens in the timer places. The property *E6* formalizes the existence of such a state:

$$\exists \text{ home state } M, \forall p \in P \text{ with } M(p) > 0 : p\bullet = \emptyset$$

Home states only exist if the batch-net is bounded. Hence we only have to analyze the batch-net for home states if it is bounded. Thus we use *transformation B* to remove timer places just as for the analysis for property *E5*. A home state, which is found in every virtual net, is a home state in the original batch-net. To calculate home states in the batch-net, we used the algorithm presented in [7]. The precondition for this algorithm is a bounded Petri-net. This is the reason why the calculation of home states follows a successful analysis for property E5 (*Bounded*).

A beneficial consequence from property *E6* is the fact, that no undesired deadlock can exist in the batch-net. Because of the definition of a home state M, marking M is reachable from any other marking of the batch-net. If only one home state exists, this home state actually is a deadlock. But since we are waiting for new impulses from the timer places, when the home state is reached, the deadlock is desired. We formalize the absence of deadlocks:

$$\forall M \in [M_0\rangle \text{ such that } M \text{ is not a home state}, \exists t \in T^* : M[t\rangle$$

The above listed six properties, which were given by the industrial partner, are the necessary ones to ensure the operation of the batch-nets in the application context. Other properties are considerable and can be integrated in our approach we present in the following.

IV. Restoring Critical Properties after Change

In this section, we show how to ensure that the run-time changes to the batch-net preserve the critical properties. A particular challenge here is, that the changes are applied at run-time, while the batch-net is executed on the server. Thus, we have to make sure that we transmit only changes to the server instance that do not violate the properties. For every critical property, we define property restoring transformations so that the changes can take place without violating the property.

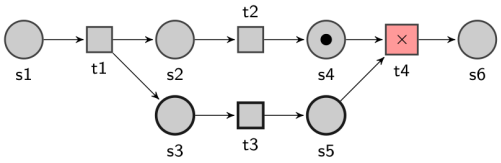
Rollback to a previous safe marking: A first measure to get the changed batch-net in a valid state, is to reset the net, so that it is executed from a previous marking, which satisfies the critical properties. For this, the firing of the transitions is stepwise reversed, according to the history of the batch-net, and the resulting markings checked for the critical properties. If a previous marking exists, so that every property is fulfilled, the marking is suggested to the user as a resolving measure. We call this process *rollback to a marking* M. Consider the example in figure 3a. In this simple batch-net, a branch is added at run-time (places and transitions with bold lines) so that transition `t4` is not able to fire any more. After the *rollback* (see figure 3b), transition `t4` can again be fired.

E1: Correct invalid timer places: If property *E1* is not fulfilled there is at least one timer place with an incoming arc. We provide a transformation, which restores this property by redirecting these arcs to a dummy place (see figure 4 for the corresponding algorithm). This ensures the correct functioning of the timer place, because the timer place is now able to create tokens at a given time. Furthermore, the redirected arcs now create tokens on the dummy-place, which can be taken into account in an analysis.

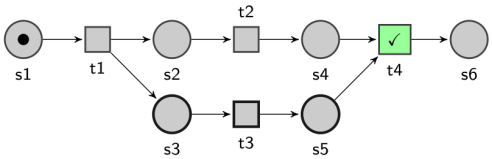
E2: Correct invalid transitions: If there is a transition without incoming arc, property *E2* is violated. To correct this violation, we provide a transformation which creates a place with a token on it and connects it with an arc to the invalid transition. The corresponding algorithm is shown in figure 5.

E3: Avoid dead transitions by creating tokens: This transformation is applied, if dead transitions are found in the changed version of the batch-net. Using a heuristic method, the transformation creates tokens on a minimal set of places, so that the dead transitions are enabled. We call these places *activating places*. For reasons of simplicity, we only regard batch-nets with arcs of capacity 1 and we do not regard conditional arcs separately in the following explanation. Furthermore, we assume that the batch-net is free of conflicts.

To motivate the algorithm which finds the minimal set of places, consider figure 6a. It is easily seen, that both `t1` and `t2` each lack a token for firing. A token in `s1` and firing of



(a) Bold places and transitions are added, `t4` is dead



(b) The firing of `t2` and `t1` was reversed, now `t4` is potentially firable

Fig. 3: Rollback to a marking M

for all $p_t \in P_T$ **with** $\exists t^* \in T^*$ **so that** $(t^*, p_t) \in F^*$**do**
  Place $p =$ **new** Place
  $P^* = P^* \cup \{p\}$
    for all $t^* \in T^*$ **with** $(t^*, p_t) \in F^*$**do**
      $F^* = F^* \cup \{(t^*, p)\}$
      $F^* = F^* \backslash \{(t^*, p_t)\}$

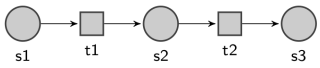
Fig. 4: Create dummy place

for all $t^* \in T^*$ **with** $\nexists p^* \in P^*$ **so that** $(p^*, t^*) \in F^*$ **do**
  Place $p =$ **new** $Place$
  $F^* = F^* \cup \{(p, t^*)\}$
  $M(p) = 1$

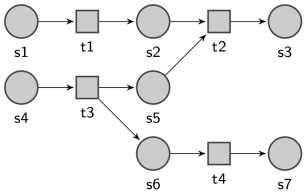
Fig. 5: Correct invalid transitions

`t1` creates the token on `s2` needed for firing `t2`. That is the reason why it suffices to create a token on `s1` to make all transitions *potentially firable*.

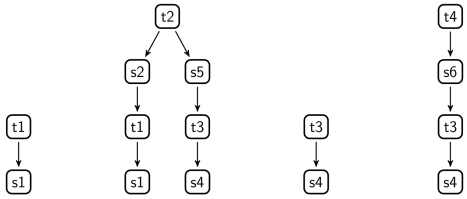
The general method to determine the minimal set thus traverses from the dead transitions to the root places. In figure 6b the situation is a bit more complicated. Here, we have to trace all dead transitions back to the reachable root places. In figure 6c, we see the reachability trees of all dead transitions.



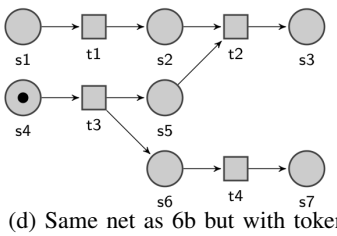
(a) Exemplary net for the marking generating algorithm



(b) Another exemplary net for the marking generating algorithm



(c) Reachability tree from dead transitions to root node in a Petri-net



(d) Same net as 6b but with token on `s4`

Fig. 6: Avoidance of dead transitions by creating tokens

Input: Petri-net PN*, set of dead transitions* T_t
Output: set of the root nods of the reachability trees
procedure CreateTraceTrees(PN, T_t)
  $W \leftarrow \emptyset$
  for all $t \in T_t$ **do**
    if $T_t \subseteq W$ **then**
      return W
    end if
    $W \leftarrow W \cup$ CreateTree$(PN, t, \emptyset)$
  end for
  return W
end procedure

Fig. 7: Computing the reachability trees

Input: Petri-net PN*, current transition* k*,* visited places P_b
Output: node of a tree
function CreateTree(PN, k, P_b)
  if $k \in P_b$ **then**
    return $\emptyset$
  end if
  if $\bullet k = \emptyset$ **then**
    return k
  else
    for all $k^* \in \bullet k$ **do**
      if $k^* \in P \wedge M(k^*) > 0$ **then**
        return $\emptyset$
      else $k^* \in P$ **then**
        $P_b \leftarrow P_b \cup k^*$
      end if
      add CreateTree(PN, k^*, P_b) as childnode to b
    end for
    return b
  end if
end function

Fig. 8: Computing single tree starting from a particular place

To identify the activating places from the reachability trees, we unite the leaves of the trees. In this case, this yields the places `s1` and `s4`.

This method works for batch-nets without loops. If the batch-net contains loops, we have to take this fact into account when computing the reachability trees. In this case, it suffices to create one token at a place within the loop.

In figure 7, 8, and 9 we give the algorithms explained above, as pseudo-code.

Since we regard batch-nets during their execution, we need to consider batch-nets in which tokens are already created, which may reduce the number of tokens still to be created.

This is the situation in figure 6d. `t3` and `t4` are now potentially firable and transition `t2` just needs a token on place `s2` to be potentially firable. This fact is taken into account in the algorithm shown in figure 8.

E4: Create self-loops to eliminate conflicts: If a place p is in conflict with several transitions, the critical property E4 is violated. Several transitions are activated, but not every transition is able to fire. We resolve this conflict by providing a transformation which integrates the affected transitions into a self-loop with the place p. In this way E4 is restored and

Input: Petri-net PN, set of dead transitions T_t
Output: set of activating places P_a
procedure DetectActivatingPlaces(PN, T_t)
$W \leftarrow \emptyset$
$P_a \leftarrow \emptyset$
$W \leftarrow CreateTraceTrees(PN, T_t)$
for $w \in W$ **do**
  $P_a \leftarrow P_a \cup$ leafs of w
end for
$P_a \leftarrow P_a \setminus$ leafs on a loop
return P_a
end procedure

Fig. 9: Determining the activating places

for all $p^* \in P^* : \left|\{t^* \in T^* | (p^*, t^*) \in F^*\}\right| > 1$ **do**
  for all such t^* **do**
    $F^* = F^* \cup \{(t^*, p^*)\}$

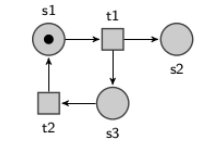
Fig. 10: Create self-loops for eliminating conflicts in batch-net

the batch-net is free of conflicts. The corresponding algorithm is shown in figure 10.

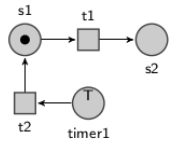
E5: Determine loops to ensure a bounded batch-net: To motivate the transformation to restore *E5*, we consider a simple example for an unbounded batch-net (figure 11a). It is easily seen that `t1`, `s3`, `t2` and `s1` form a loop. `s2` is an outgoing place of transition `t1` which is located in the loop. In this way, an infinite set of tokens can be created on the place `s2`. We consider this in the following definition and theorem.

Definition 7. *Given a Petri-net P, a simple loop $L = (S', T')$ is a subset $S' \subseteq S$ such as $T' \subseteq T$ with $\forall t \in T' : \left|\bullet t\right| = 1 \wedge \bullet t \subseteq S'$ and $\forall s \in S' : \exists t \in \bullet s : t \subseteq T'$.*

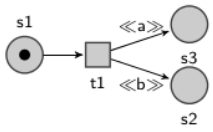
Because every simple loop L can iterate infinitely, an infinite



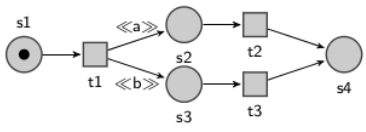
(a) Simple loop



(b) Broken up simple loop



(c) Example of home state preventing conditional arc



(d) Reunification of the arcs leads to home state

Fig. 11: Examples for restoring critical properties

Input: Batch-net B
Output: Set of all loops W_p
procedure DETERMINEALLLOOPS(B)
  visited places $P_v \leftarrow \emptyset$
  temporal set of loops $W_p^* \leftarrow \emptyset$
  $W_p \leftarrow \emptyset$
  while $P_v \neq P$ **do**
    $p \leftarrow$ random place in $P \setminus P_v$
    $W_p^t \leftarrow$ DETERMINELOOPS$(B, \emptyset, p)$
    if $W_p^t \neq \emptyset$ **then**
      $W_p \leftarrow W_p \cup W_p^*$
      $P_v \leftarrow P_v \cup$ all places in W_p^*
    else
      $P_v \leftarrow P_v \cup p$
    end if
  end while
  return W_p
end procedure

Input: Batch-net B, current path w, current place p
Output: Set of loops depending on w and p
procedure DETERMINELOOPS(B, w, p)
  if $s \in w$ **then**
    return path from p to the last place of path w
  end if
  if $|\bullet p| = 0$ **then**
    return $\emptyset$
  end if
  for all $t \in \bullet p$ **do**
    if $|\bullet t| > 1$ **then**
      return $\emptyset$
    else
      $w \leftarrow$ extend w to $\bullet t$
      return $W_s \cup$ DETERMINELOOPS(B, w, p)
    end if
  end for
end procedure

Fig. 12: Determining loops that have to be broken up

number of tokens can be generated on every place leaving the loop. That is what is formalized in the following theorem.

Theorem 1. *Given a Petri-net P such as a simple loop $L = (S', T')$. Then in every place $s \in S \setminus S'$ with $\exists t \in \bullet s : t \in T'$, an unbounded number of tokens can be created.*

The algorithm, which is based on this theorem, iterates over the places till for every place it is determined, whether it belongs to a loop or not. If all loops are discovered, every loop will be broken up in a random place. For this, the incoming arc of this place is deleted. The randomly chosen place is transformed into a timer place, which is provided with a random time interval. In this way, an unbounded number of tokens can still be created on the outgoing place. We remark, that the batch-net created this way is not necessarily semantically identical. In the implementation of this countermeasure, the user will be informed about induced semantic changes. For the above example, the broken up loop is shown in figure 11b. In figure 12, we give the algorithm to determine loops that have to be broken up.

E6: Determine home state preventing conditional arcs: In a valid batch-net, conditional arcs are the only possibility

to split the control flow. Accordingly, the conditional arcs may prevent the existence of a home state. To illustrate this fact, we consider the net shown in figure 11c. It is easily seen, that the Petri-net has no home state, because the token on `s1` moves, depending on the condition, either to `s2` or to `s3`. The problem can be resolved by bringing the two branches together again (see figure 11d). This transformation is applicable to Petri-nets where the subnets of the conditional arcs are a sequence of places and transitions connected each by one arc.

V. Validation of the approach

We now demonstrate how we implemented the run-time change to the batch-net. For this, the algorithms for generation of conflict resolving measures are embedded in the application context. They are used in the context of the *BatchNet editor* which supports change of batch-net instances at run-time. In the following, we first present the general structure of the workflow of the tool for an overview. Then we introduce the underlying technical components of the application and conclude with some implementation details about the analysis-process.

Functional requirements: Once the tool is started, the user connects to an application server. On this server, an instance of the batch-net is hosted. When connected, the batch-net located on the server is imported into the tool and instantiated with the current marking. Now the user can make changes to the local copy of the batch-net. If the user wants to transmit the changed net to the server, it is validated with respect to the properties introduced in section III. If the analysis-process is successful, the changes are transmitted immediately. If the analysis-process is not successful, possible conflict resolving measures are presented to the user, as explained in section IV. The measures can then be applied or be discarded by the user. The validation check of the net distinguishes validity with respect to the initial marking and to the current marking. For the first case, the changes are stored to the underlying XML-file. For the second case, the changes are done to the run-time instance on the server.

Technical components: To illustrate the implementation in a more fine grained way, we show how the implementation is subdivided into several components and how they relate to each other (compare figure 13). The user of the tool interacts with the *BatchNet editor*-component. The *BatchNet editor* realizes the change to the local copy of the batch-net. It is synchronized with the *Quasar application*-component via the interfaces *Read net* and *Write net* of *Quasar application*. In this way the batch-net instance with the current marking can be imported into the *BatchNet editor* and validated changes can be written down to the running instance in the *Quasar application*. The changes made by the user are validated by the *Validation algorithms*-component which provides the *Validate net* interface to the *BatchNet editor*. To analyze the batch-net instances, it is necessary to transform the regarded batch-nets to corresponding Petri-nets as explained in section II. Therefore the *PetriNet model*-component provides a *Convert from BatchNet* interface, so that *Validation algorithms* works

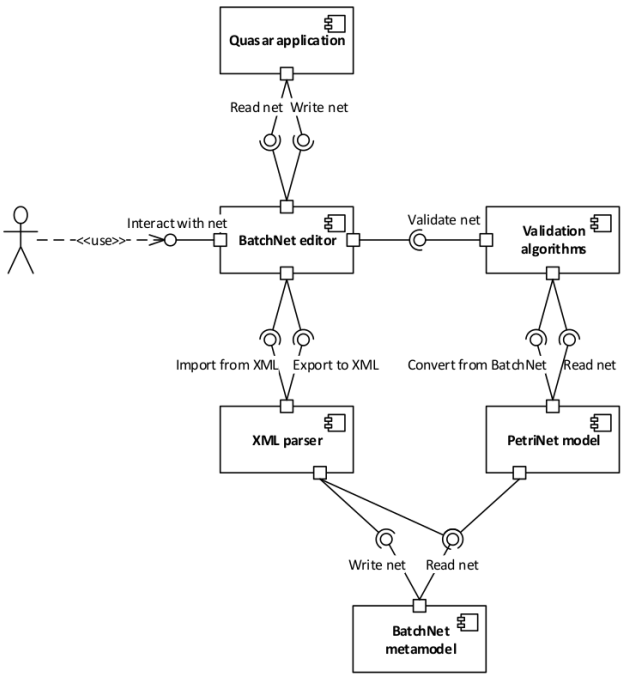


Fig. 13: Architecture of the tool shown as a UML-component diagram

primarily on Petri-nets. The batch-net, which is shown to the user in the *BatchNet editor*, is exported in the background to an XML-file according to the *BatchNet metamodel*. The other way around, the batch-net instance can be filled with an XML-file in a proprietary format.

Implementation details: The analysis-process of the changed batch-net is embedded in three abstract layers. Namely *analyzer* layer, *comparer layer* and *feature layer*. The analyzer layer consists of the interface `Analyzable`, its abstract implementation `AbstractAnalyzer` and the concrete implementations of the abstract class. `Analyzable` defines only the method `Analyzable.analyze()` as entry point and returns a `Result` object which contains information regarding the analysis. In `AbstractAnalyzer`, the abstract implementation of `Analyzable`, the analysis-process is divided into three steps. First of all the batch-net is transformed in an ordinary Petri-net by the class `BatchNetTransformation`. Several Petri-nets originate if and only if the batch-net contains *conditional arcs*. How batch-nets are transformed into ordinary Petri-nets is defined in section II. For each of the converted Petri-nets the method `AbstractAnalyzer.analyzeSingleNet(PetriNet net)` is called. This method is abstract, this means that every concrete subclass of `AbstractAnalyzer` has to fill this method with code. This code is the implementation of the analysis-processes given in section III. At last, the results of the analysis-processes of the individual nets need to be merged into one result valid for the original net. This happens in the method `AbstractAnalyzer.aggregateResults()`. The method per property is discussed in section III.
The `comparer layer` is necessary to compare

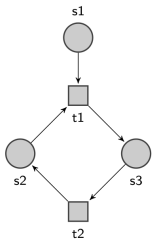


Fig. 14: A net with a trap

the results of the analysis for the initial and the current marking. The basis for this is the class `AbstractComparer` which is connected with two objects of type `Result`. One for the initial marking and one for the current one. Important in this class is the method `AbstractComparer.getResult(Current-PetriNet, boolean useInitialMarking)`. It is implemented by the subclasses and returns the result of the respective analysis with the initial and the current marking.

The *feature layer* is responsible for the interconnection of the actual analysis-process and the user interface. We do not consider it in this paper.

Evaluation: While we tested our approach on several batch-nets, it turned out that it does not work optimal for batch-nets with traps. A trap is a subset of places $P^{'} \subseteq P$ with $P^{'}\bullet \subseteq \bullet P^{'}$. Lucidly it is a part of the net where tokens get in but not out (compare figure 14). The places `s1` and `s2` form a trap. If the reachability trees are build from `t1` or `t2` respectively, `t1-s3-t2-s2` is discovered as a loop. Accordingly on `s2` or `s3` a token is created as a conflict resolving measure. Additionally `s1` is recognized as activating place. As a consequence the token on `s2` or `s3` is not needed and thus our method does not compute an optimal solution.

Practical validation: Our approach was tested and implemented at Capgemini, a large consulting and IT-service provider. Part of the IT-services at Capgemini is the development of individual software and package-based solutions within the department of *custom solutions development*. In this department the TECBASE was introduced, which uses the concepts and services for software-architectures provided by Quasar [5]. Quasar itself is mainly used in a project for a logistics-service provider. The batch-component, we presented in this paper, is in use in this application context to model batch-dependencies based on Petri-nets. This shows that our approach is not just a theoretical concept but a serious technique used in productive invironments.

Run-time of the analysis-process: We tested the implementation of the analysis-process of our approach with several randomly generated Petri-nets. We measured the time, needed to compute the countermeasures in dependence of the number of nodes N and the maximum number of tokens T of a single place in the Petri-net. Each value listed is the average of ten test-runs for each configuration of N and T. The configuration of N and T was chosen according to comparable sizes coming from real batch-job applications. The Petri-nets were generated according to the following scheme:

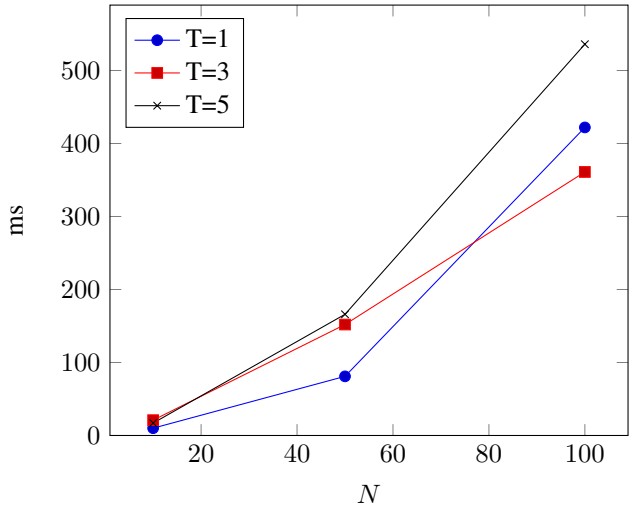


Fig. 15: Measured run-time in dependence of the number of nodes $N \in \{10, 50, 100\}$ in the Petri-net

- Every node in the net is randomly chosen to be a place or a transition.
- Every place gets either no arc or one arc to a random transition.
- Every transition gets 0 to $(N - \#\text{transitions})$ arcs to randomly chosen places.
- On every place up to T tokens are created randomly.

The measured run-time in milliseconds is shown in figure 15. In figure 15, we fixed the number of tokens T for each graph, to show the dependence of the run-time from the number N of nodes in the Petri-net. By increasing the number of nodes in the net by an order of magnitude comparable to real batch-job applications, the run-time rises significantly. The application of the analyses-process showed, that the run-time of our approach is short enough to make it applicable to real batch-nets.

VI. Related Work

Similar to our concept of batch-nets, Hanisch defined Petri-nets with timed arcs in context of batch process control [8], [9]. In contrast to our approach, he defines timed arcs and not timed places. The behavioral outcome of this concept is the same, compared to our definition, but he does not extend Petri-nets by means of conditional arcs and special batch transitions. The emphasis lies in the analysis of the extended Petri-net concept rather than the change at run-time. Chang et al. [10] map UML activity diagrams with time properties to timed coloured Petri-nets to model and evaluate real-time systems.

The survey [11] presents modeling techniques for Petri-nets in different contexts and quantitative and qualitative analysis of Petri-nets applied to batch processes, as well as supervisory and coordinate control and planning and scheduling.

Andreu et al. deal with the problem to concurrent operate with continuous and discrete models [12]. They analyze how the hierarchical approach, used in discrete manufacturing systems, can be extended to batch systems. An event generator is utilized to guarantee consistency between the continuous model of the process and the discrete model of the plant.

Ghaeli et al. present a heuristic search algorithm for short-term scheduling of batch plants based on the reachability tree in timed Petri-nets [13]. Van der Aalst discusses the use of Petri-nets in the context of workflow management and introduces workflow management as an application domain for Petri-nets. He presents results with respect to the verification of workflows and highlights some Petri-net based workflow tools [14], [15]. Lloyd and Salleh proposed a design scheme of the batch process plant modeled by timed Petri-nets [16].

Wrt. change at run-time, [17] considers the dynamic change bug to support workflow change with an approach for computing a safe change region in a workflow, a specialization of Petri-nets. This work does not regard the change of a net which is currently running. Llorens and Oliver utilize reconfigurable Petri-nets to model and verify dynamic changes to concurrent systems [18]. A configuration of a software system is represented by such a reconfigurable Petri-net and a mechanism is presented which enables the system to change from one configuration to another. Oreizy et al. present an architecture-based approach for run-time software system reconfiguration, highlighting the beneficial role of architectural styles and software connectors [19].

Compared to the above mentioned approaches, the emphasis of our approach lies on the change to a batch-net at run-time, which has not yet been considered by these works.

VII. Conclusion

In many application areas, e.g. for logistics-service providers, competitive pressure leads to the necessity to respond more quickly to changed conditions. That is the reason why changes to a running instance of a batch-job net had to be made possible. In this paper we showed that it is feasible to change the running instance of a batch-net, without introducing unintentional structures. Because our approach makes use of the concept of Petri-nets, it can not easily be generalized to other domains using other concepts. In this paper we presented six critical properties a batch-net has to fulfill at run-time. For each property, we presented an algorithm that checks for the property. The critical properties we derived from domain needs are preserved under change by suggesting countermeasures to the user in case of an invalid batch-net. Because our approach was implemented in a real industrial environment, we ensured that it is not just a theoretical concept but a valuable contribution to facilitate the organization of batch jobs. To compute the virtual nets, to analyze batch-nets with conditional arcs, is very expensive, because it has to happen recursively. But the batch-nets coming from application context are not so big so that it will be a serious problem for the computing time or computing resources. Since the heuristic method to determine the minimal set of enabling places does not lead to the optimal solution in every case further research in this area is necessary. For further future work, we are planning to apply this work within the domain of security-critical information systems [20] by focussing on secure information flow properties [21], and to create a link to the source code level using techniques from program comprehension [22].

Acknowledgement: This research is funded by the DFG project SecVolution (JU 2734/2-1 and SCHN 1072/4-1) which is part of the priority programme SPP 1593 “Design For Future - Managed Software Evolution”.

***Index Terms*—petri-net; batch-net; run-time change; evolution;**

References

[1] P. Oreizy and R. N. Taylor, “On the role of software architectures in runtime system reconfiguration.” in *IEE Proceedings - Software*, 1998, pp. 137–145.

[2] IBM Corporation, “Mainframes working after hours: Batch processing,” http://www-01.ibm.com/support/knowledgecenter/zosbasics/com.ibm.zos.zmainframe/zconc_batchproc.htm, 2014/11/20.

[3] ORACLE, “Oracle SOA Suite and The Modernization of Job Scheduling,” http://www.oracle.com/technetwork/topics/modernization/uc4.pdf, 2014/09/30.

[4] T. Murata, “Petri Nets: Properties, Analysis and Applications,” *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[5] G. Engels, M. Kremer, T. Nötzold, T. Wolf, K. Prott, J. Hohwiller, A. Hofmann, A. Seidl, D. Schlegel, and O. Nandico, “Quasar 3.0 - A situational approach to Software Engineering,” 2012.

[6] M. S. Keller, “Take command: cron: Job Scheduler,” *Linux Journal*, vol. 1999, no. 65es, 1999.

[7] P. Wang, Z. Ding, and H. Chai, “An algorithm for generating home states of Petri-Nets,” *Journal of Computational Information Systems*, vol. 7, no. 12, pp. 4225–4232, 2011.

[8] H.-M. Hanisch, “Analysis of Place/Transition Nets with Timed Arcs and its Application to Batch Process Control,” in *Application and Theory of Petri-nets*, ser. LNCS. Springer, 1993, vol. 691, pp. 282–299.

[9] ——, “Coordination Control Modelling in Batch Production Systems by Means of Petri-nets,” *Computers & chemical engineering*, vol. 16, no. 1, pp. 1–10, 1992.

[10] X. Chang, L. Huang, J. Hu, C. Li, and B. Cao, “Transformation from activity diagrams with time properties to Timed Coloured Petri Nets,” in *Computer Software and Applications Conference (COMPSAC), 2014 IEEE 38th Annual*, July 2014, pp. 267–272.

[11] T. Gu and P. A. Bahri, “A survey of Petri net applications in batch processes,” *Computers in Industry*, vol. 47, no. 1, pp. 99–111, 2002.

[12] D. Andreu, J.-C. Pascal, H. Pingaud, and R. Valette, “Batch Process Modelling Using Petri Nets,” in *1994 IEEE International Conference on Systems, Man, and Cybernetics, 1994. Humans, Information and Technology.*, vol. 1, Oct 1994, pp. 314–319 vol.1.

[13] M. Ghaeli, P. A. Bahri, P. Lee, and T. Gu, “Petri-Net based formulation and algorithm for short-term scheduling of batch plants,” *Computers & chemical engineering*, vol. 29, no. 2, pp. 249–259, 2005.

[14] W. M. van der Aalst, “The Application of Petri-nets to Workflow Management,” *Journal of circuits, systems, and computers*, vol. 8, no. 01, pp. 21–66, 1998.

[15] W. M. Van Der Aalst, “Workflow Verification: Finding Control-Flow Errors using Petri-net based Techniques,” in *Business Process Management*. Springer, 2000, pp. 161–183.

[16] S. Lloyd and Y. M. Salleh, “Modeling and control design of batch process plant by Timed Petri-net,” in *30th IEEE Conference on Decision and Control*. IEEE, 1991, pp. 1531–1536.

[17] W. M. van der Aalst, “Exterminating the Dynamic Change Bug: A concrete Approach to support Workflow Change,” *Information Systems Frontiers*, vol. 3, no. 3, pp. 297–317, 2001.

[18] M. Llorens and J. Oliver, “Structural and Dynamic Changes in Concurrent Systems: Reconfigurable Petri-nets,” *Computers, IEEE Transactions on*, vol. 53, no. 9, pp. 1147–1158, 2004.

[19] P. Oreizy, N. Medvidovic, and R. N. Taylor, “Runtime Software Adaptation: Framework, Approaches, and Styles,” in *Companion of ICSE*. ACM, 2008, pp. 899–910.

[20] E. Fernández-Medina, J. Jürjens, J. Trujillo, and S. Jajodia, “Model-driven development for secure information systems,” *Information & Software Technology*, vol. 51, no. 5, pp. 809–814, 2009.

[21] J. Jürjens, “Secure information flow for concurrent processes,” in *CONCUR 2000*, ser. LNCS, vol. 1877. Springer, 2000, pp. 395–409.

[22] D. Ratiu, M. Feilkas, and J. Jürjens, “Extracting domain ontologies from domain specific APIs,” in *12th European Conference on Software Maintenance and Reengineering (CSMR 08)*, 2008, pp. 203–212.