Context-Aware Process Injection
Enhancing Process Flexibility by Late Extension of Process Instances

Nicolas Mundbrod(✉), Gregor Grambow, Jens Kolb, and Manfred Reichert

Institute of Databases and Information Systems, Ulm University, Ulm, Germany
{nicolas.mundbrod,gregor.grambow,jens.kolb,manfred.reichert}@uni-ulm.de
http://www.uni-ulm.de/dbis

Abstract. Companies must cope with high process variability and a strong demand for process flexibility due to customer expectations, product variability, and an abundance of regulations. Accordingly, numerous business process variants need to be supported depending on a multiplicity of influencing factors, e.g., customer requests, resource availability, compliance rules, or process data. In particular, even running processes should be adjustable to respond to contextual changes, new regulations, or emerging customer requests. This paper introduces the approach of context-aware process injection. It enables the sophisticated modeling of a context-aware injection of process fragments into a base process at design time, as well as the dynamic execution of the specified processes at run time. Therefore, the context-aware injection even considers dynamic wiring of data flow. To demonstrate the feasibility and benefits of the approach, a case study was conducted based on a proof-of-concept prototype developed with the help of an existing adaptive process management technology. Overall, context-aware process injection facilitates the specification of varying processes and provides high process flexibility at run time as well.

Keywords: Process injection · Process flexibility · Process variability · Process adaptation · Data collection processes

1 Introduction

In today's globalized world, companies face various challenges like increased customer expectations, complex products and services, demanding regulations in different countries, or fulfillment of social responsibility. As a result, companies need to cope with high process variability as well as a strong demand for process flexibility. This means that in many of their business processes the course of action is influenced by an abundance of *process parameters* like external context factors, intermediate results, and process-related events (e.g., successful termination of process steps). Consequently, ordinary process models comprise complex decisions allowing for various alternative courses of actions as well as

© Springer International Publishing Switzerland 2015
C. Debruyne et al. (Eds.): OTM 2015 Conferences, LNCS 9415, pp. 127–145, 2015.
DOI: 10.1007/978-3-319-26148-5_8

interdependencies among these decisions that are hardly comprehensible for process modelers. In addition, an automated, controlled and sound adaptation of (long-running) processes instances is required to address contextual changes, new regulations, or emerging customer requests at run-time.

The complex development, production, or reporting processes in the automotive and electronics industry may be regarded as valuable examples [8,12,18]. Typically, these processes rely on the companies' sensitive supply chains. Hence, business partners and diverse activities have to be incorporated dynamically on demand. The following application scenario (cf. Fig. 1), we derived in the context of a case study, illustrates the complexity and dynamics of such processes.

Application Scenario: ***Data Collection Processes***
Due to regulations, an automotive manufacturer needs to provide sustainability information. In particular, sustainability indicators relating to its production are requested: one indicator deals with the REACH[a] compliance of the entire company, another one addresses the greenhouse gas emissions during the production of a certain product. To gather the data, process *Data Collection 1* is deployed to request a REACH compliance statement from a supplier. Additionally, two other suppliers must be contacted to report the greenhouse gas emissions (process *Data Collection 2*). While both data collection processes have activities in common, many activities are specifically selected for each process. A request regarding REACH compliance, e.g., implies a legally binding statement and, thus, a designated representative must sign the data. However, if the CEO was not available, activity *Sign Data* can be delayed or skipped.

[a] Regulation (EC) No 1907/2006: Registration, Evaluation, Authorisation and Restriction of Chemicals

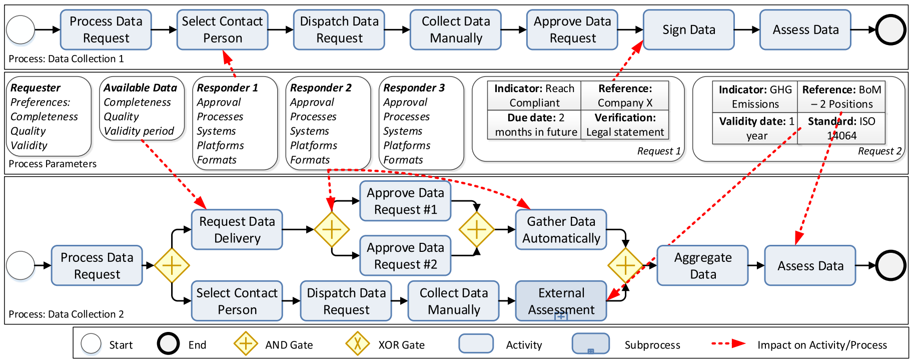


Fig. 1. Application Scenario with two Data Collection Processes

To systematically support long-running and varying processes that require (data-driven) run-time flexibility, we introduce the approach of *context-aware process injection* (CaPI). Taking the current context of a process into account, CaPI enables the controlled, but late injection (i.e., insertion) of process fragments into a lean base process. Using so-called *extension areas*, the correctness of the process' control and data flow is ensured after injecting process fragments at run time. The feasibility of CaPI is demonstrated by implementing a proof-of-concept prototype based on existing adaptive process management technology.

Underpinning our research, we applied the *design science* research methodology [15]. In particular, our work can be categorized as a *design- and development-centered approach* accordingly. Based on an analysis of application scenarios (e.g., [8,11,17]) and process-related backgrounds (cf. Sect. 2) as well as the evaluation of existing approaches (cf. Sect. 7), we iteratively elaborated the CaPI approach (cf. Sect. 3). The latter comprises the specification of its components (cf. Sect. 4). Furthermore, we give insights into the *process of context-aware process injection* (cf. Sect. 5). To validate the approach, a poof-of-concept prototype was developed enabling the usage and evaluation in different application scenarios (cf. Sect. 6). Finally, Section 8 concludes the paper giving a summary and an outlook.

2 Backgrounds

To make CaPI applicable to existing activity-centric process modeling notations, it relies on the process model definition given in Def. 1.

Definition 1. *A* ***process model*** PM *is a tuple* (N, E, NT, ET, EC) *where:*

- N *is a set of process nodes and* $E \subseteq N \times N$ *is a precedence relation (directed edges) connecting process nodes,*
- $NT : N \rightarrow \{\mathtt{Start}, \mathtt{End}, \mathtt{Activity}, \mathtt{ANDsplit}, \mathtt{ANDjoin}, \mathtt{ORsplit}, \mathtt{ORjoin}, \mathtt{XORsplit}, \mathtt{XORjoin}, \mathtt{DataObj}\}$ *assign to each* $n \in N$ *a node type* $NT(n)$*;* N *is divided into disjoint sets of start/end nodes* C $(NT(n) \in \{\mathtt{Start}, \mathtt{End}\})$*, activities* A $(NT(n) = \mathtt{Activity})$*, gateways* G $(NT(n) \in \{\mathtt{ANDsplit}, \mathtt{ANDjoin}, \mathtt{ORsplit}, \mathtt{ORjoin}, \mathtt{XORsplit}, \mathtt{XORjoin}\})$*, and data objects* D $(NT(n) = \mathtt{DataObj})$*,*
- $ET : E \rightarrow \{\mathtt{ControlEdge}, \mathtt{LoopEdge}, \mathtt{DataEdge}\}$ *assigns a type* $ET(e)$ *to each edge* $e \in E$*,*
- $EC : E \rightarrow Conds \cup \{True\}$ *assigns a transition condition or true to each control edge* $e \in E, ET(e) \in \{\mathtt{ControlEdge}, \mathtt{LoopEdge}\}$*.*

Note that we take sound process models for granted, i.e., a process model has one start (no incoming edges) and one end node (no outgoing edges) [17]. Further, the process model has to be *connected*, i.e., each activity can be reached from the start node, and from each activity the end node is reachable. Data consumed (delivered) as input (output) by the process model is written (read) by the start (end) node. Finally, branches may be arbitrarily nested, but must be safe (e.g., a branch following an `XORsplit` must not merge with an `ANDjoin`). Due to lack of space, we refer to literature for a detailed look on process model soundness [19]. Def. 2 introduces the notion of a *SESE (Single Entry Single Exit)* fragment:

Definition 2. *Let $PM := (N, E, NT, ET, EC)$ be a process model and $N' \subseteq N$ be a subset of activities. The subordinated process model PM' induced by N' and their corresponding edges $E' \subseteq E$ is denoted as* ***Single Entry Single Exit*** *(SESE) fragment iff PM' is connected and has exactly one incoming and one outgoing edge connecting it with PM. If PM' has no preceding (succeeding) nodes, PM' has only one outgoing (incoming) edge.*

Based on a process model PM, a process instance PI may be created, deployed and executed at run time. Def. 3 defines a process instance formally:

Definition 3. *A* ***process instance*** *PI is defined as a tuple (PM, NS, Π) where:*

- *$PM := (N, E, NT, ET, EC)$ denotes the process model PI is executed on,*
- *$NS : N \rightarrow \{\mathtt{NotActivated}, \mathtt{Activated}, \mathtt{Running}, \mathtt{Skipped}, \mathtt{Completed}\}$ describes the execution state of each node $n \in N$ with $NT(n) \neq \mathtt{DataObj}$,*
- *$\Pi := \langle e_1, \ldots, e_n \rangle$ denotes the current execution trace of PI where each entry e_k is related either to the start or completion of an activity.*

3 Context-Aware Process Injection in a Nutshell

The key objective of CaPI is to ease the sophisticated modeling of process variants at design time and to enable the automated, controlled adaption of processes at run time. Therefore, the central entity of CaPI is the *context-aware process family* (CPF) (cf. Fig. 2). In detail, a CPF comprises a *base process* model with *extension areas* (cf. Sect. 4.1), *contextual situations* (cf. Sect. 4.3) based

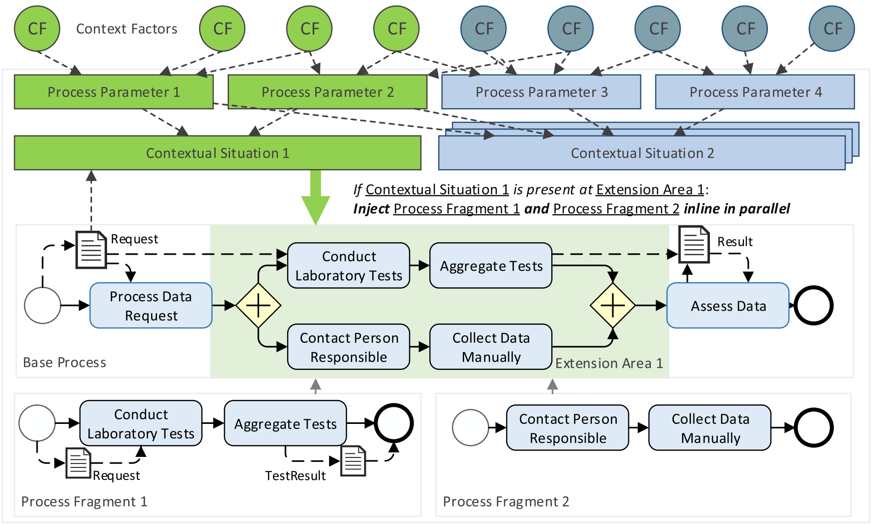


Fig. 2. Overview of a Context-Aware Process Family

on *process parameters* (cf. Sect. 4.2), a set of *process fragments* injected at the extension areas at run time, and a set of *injection specifications* (cf. Sect. 4.4).

Establishing the *separation of concerns* principle for modeling process variants, the *base process model* solely contains decisions and activities shared by all variants of the process, known at build time, and not being changed at run time. By contrast, *extension areas* represent the dynamic part of the process. Hence, a process modeler may focus on modeling predictable activities first to add the dynamic parts of the base process model subsequently. In particular, extension areas are used to automatically inject process fragments into the base process at run time based on the present *contextual situation* as well as on well-defined *injection specifications*. An extension area allows for the dynamic injection of any number of parallel process fragments. In turn, contextual situations are defined through conditions expressed in first-order logic taking *process parameters* and even data objects of the base process model into account. In this context, process parameters are connected to dynamic, external factors influencing the process injection's decision making. While injecting process fragments, CaPI takes care of correct data flow mappings as well: data objects of an injected process fragment are automatically connected to existing ones of the base process.

By this means, CaPI enables controlled, but dynamic configurations and changes of long-running and varying processes at run time. Through relying on insertions of process fragments solely, CaPI allows process modelers to increasingly focus on the particular variants instead of struggling with a highly complex process model capturing all variants.[1] Furthermore, process modelers may directly integrate contextual influences into the modeling of variants as complex external context factors are abstracted by meaningful process parameters and reusable contextual situations. In turn, CaPI is able to cope with contextual run-time changes through the late evaluation of contextual situations at given extension areas to finally inject the proper process fragments. Thereby, the automated and consistent construction of data flow between the injected process fragments and the underlying base process mitigates the efforts of involved users. Further, it empowers process activities to seamlessly read and write data.

Before presenting the key components of a CPF and CaPI, Def. 4 formally specifies the concept of a *context-aware process family* (CPF). Note that a process fragment may be the base process of another CPF and, thus, modularization can be achieved as well (recursive nesting is disallowed).

Definition 4. *A* ***context-aware process family*** *is defined as a tuple* $CPF = (BP, EA, PP, CS, PF, IS)$ *where:*

- BP *is the* base process model*,*
- EA *is a set of* extension areas *in the* BP*,*
- PP *is a set of* process parameters*,*
- CS *is a set of* contextual situations*,*

[1] Note that other kind of dynamic changes, like deleting or moving activities, may be also introduced by authorized users based on the features of the adaptive process management technology [6] used.

– *PF is a set of* process fragments*; each process fragment is a process model,*
– *IS is a set of* injection specifications*.*

4 Components of Context-Aware Process Families

4.1 Extension Areas

In order to enable the controlled extension of processes at run time, extension areas are introduced representing the dynamic part of a CPF. Based on the current contextual situation, process fragments may be dynamically injected into extension areas at run time. More precisely, an extension area is defined by two extension points—each referring to a node (i.e. start/end nodes, activities, gateways) of the base process model (cf. Fig. 3). If the nodes referenced by the extension points directly precede each other, a process fragment can be easily injected into the base process. If some nodes exist in between, a process fragment may be injected among these nodes (cf. Sect. 4.4) or, alternatively, gateways may be employed to insert the process fragment in parallel. The different possibilities of injecting process fragments are discussed in Section 5. Def. 5 formally describes extension areas and posits constraints to ensure that the injection of process fragments into a base process BP always leads to a modified, but still sound process BP'. In this context, overlaps of extension areas may result in problems regarding the concurrent injection of process fragments (cf. Sect. 5.2).

Definition 5. *Let $CPF = (BP, EA, PP, CS, PF, IS)$ be a context-aware process family and $BP = (N, E, NT, ET, EC)$ be the base process. Every **extension area** $ea \in EA$ is described by a set of two extension points $\{EP_s, EP_e\} \subseteq N \times \{\texttt{Pre}, \texttt{Post}\}$ where:*

– *Every extension point $EP_x = (n_x, scope), x \in \{s, e\}$ refers to corresponding nodes $n_x \in N, NT(n_x) \neq \texttt{DataObj}$ in BP and additionally exposes a scope; the latter determines whether ea starts (ends) just before ($scope = \texttt{Pre}$) or directly after ($scope = \texttt{Post}$) the referenced node n_x,*
– *EP_s (EP_e) may only refer to the scope $\texttt{Post}$ ($\texttt{Pre}$) of the start (end) node of BP; EP_s (EP_e) must not refer to the end (start) node of BP,*
– *The referenced nodes $n_s, n_e \in N$ embrace a subordinated process model PM' induced by a subset of activities $N' \subseteq N$ and respective edges $E' \subseteq E$; PM' always corresponds a SESE fragment and must not contain any other extension areas starting (ending), but not ending (starting) in PM' (nesting of extension areas is allowed, but no overlaps).*

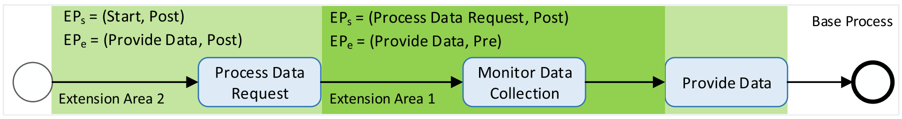


Fig. 3. Examples of Extension Areas in a Base Process

4.2 Process Parameters

Typically, long-running and varying processes are influenced by context factors, e.g., the number of involved parties or the availability of data. To include such context factors into the decision making regarding the injection of process fragments and hence the concrete course of action of the overall process, we utilize a predefined set of *process parameters* (cf. Def. 6). The set of process parameters additionally enables the exchange of entire CPFs between application scenarios as it abstracts from a concrete set of context factors (i.e., only a mapping between the context factors and process parameters need to be conducted again).

Definition 6. *Let $CPF = (BP, EA, PP, CS, PF, IS)$ be a context-aware process family. A* ***process parameter*** *$pp \in PP$ is a tuple $(ppDefault, ppValue, ppDom)$ where:*

- *$ppDefault \in PPDom$ is an optional default value of the process parameter,*
- *$ppValue \in PPDom$ is the current value of the process parameter,*
- *$ppDom \subseteq Dom$ is the domain of pp with Dom denoting the set of all atomic domains (e.g., String, Integer)*

Note that value $ppValue$ of process parameter pp is set by a context mapping (component) at run time (cf. Sect. 5.2). To focus on the controlled process adaption, we rely on simple rule-based mapping for context factors (cf. Fig. 4).

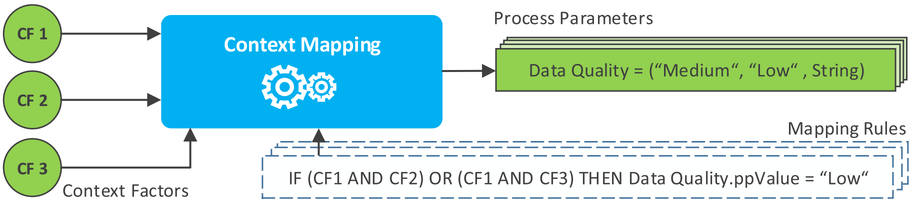

CF 1
CF 2
CF 3
Context Factors
Context Mapping
Process Parameters
Data Quality = (“Medium“, “Low“ , String)
Mapping Rules
IF (CF1 AND CF2) OR (CF1 AND CF3) THEN Data Quality.ppValue = “Low“

Fig. 4. Illustrative Mapping of Context Factors on Process Parameters

Consequently, process parameters may be also leveraged to provide meta information regarding the current execution trace (cf. Def. 3) or the process fragments injected at run time. Such process parameters can then be used to model interdependencies among contextual situations and process fragments, respectively. Finally, a process parameter may have compound values (e.g., sets, lists) as well—however, we omit a formal definition of complex parameters here.

4.3 Contextual Situations

A specific process variant may rely on several occurring *contextual situations*, which are based on the combination of various process parameters and, especially, their current values. For example, a company may insist on a four-eyes-principle approval process in case data is intended for a specific customer group or relates to a specific regulation. Hence, the same contextual situations may be leveraged at different extension areas to inject process fragments. Based on this observation, contextual situations (cf. Fig. 5) are defined by conditions expressed in a first-order logic relying on the set of process parameters (cf. Sect. 4.2) and data objects of the base process (cf. Def. 7). As opposed to traditional modeling of business processes, we enable the integration of external context factors as well as reutilization of contextual situations across the process model. As default process parameters may provide meta information regarding the current execution trace or the process fragments injected at run time, interdependencies can be modeled in contextual situations correspondingly.

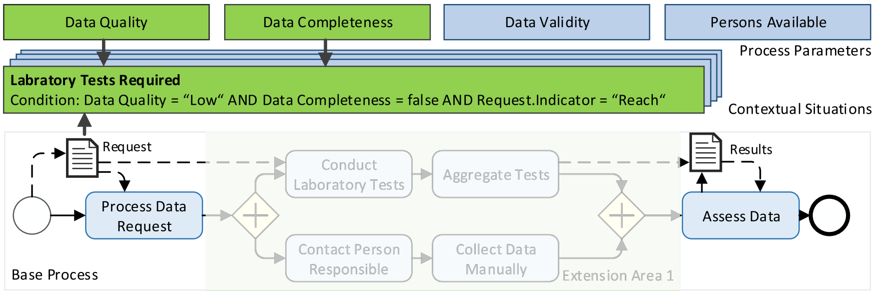


Fig. 5. Contextual Situation based on Process Parameters and Data Objects

Definition 7. *Let $CPF = (BP, EA, PP, CS, PF, IS)$ be a context-aware process family and $BP = (N, E, NT, ET, EC)$ be the corresponding base process. A* ***contextual situation*** *$cs \in CS$ is defined by a condition expressed in first-order logic. For every predicate $par_k\ \theta\ val_k, \theta \in \{``=“, `` \leq “, \ldots\}, val_k \in Dom(par_k)$ of the condition, par_k either corresponds to a process parameter ($par_k \in PP$) or a data object ($par_k \in N, NT(par_k) = DataObj$).*

4.4 Injection Specifications

Finally, *injection specifications* determine the injection of a process fragment to an extension area in a given contextual situation (cf. Def. 5). To ensure data flow correctness after the injection, in addition, the mapping of data elements is considered in the injection specifications. Especially, this includes a mapping of required input and output data objects of the process fragment (or, to be more precise, of their activities) to the existing data objects of the base process. This mapping may be even extended to data objects of other process fragments, which are supposed to be injected in the base process as well (cf. Sect. 5).

Definition 8. *Let $CPF = (BP, EA, PP, CS, PF, IS)$ be a context-aware process family and $BP = (N, E, NT, ET, EC)$ be the corresponding base process. An* ***injection specification*** *$is \in IS$ corresponds to a tuple $(EA_{IS}, CS_{IS}, PF_{IS}, InjType, InjPattern, InjRate, InjTrigger, InjRank, DR, DW)$ where:*

- *$EA_{IS} \in EA$ is a specific extension area, $CS_{IS} \in CS$ a specific contextual situation, and $PF_{IS} \in PF$ a specific process fragment,*

- $InjType := \{\texttt{Inline}, \texttt{Sub-process}\}$ *is the injection type denoting whether* PF_{IS} *is injected inline or as a sub-process,*
- $InjPattern := \{\texttt{Parallel}, \texttt{Sequential}\}$ *is the injection pattern denoting whether* PF_{IS} *is injected in parallel to the existing control flow between the extension points of* EA_{IS} *or sequentially into the existing control flow,*
- $InjRate := \{\texttt{Single}, (\texttt{Multiple}, fre)\}$ *is the injection rate denoting whether* PF_{IS} *is injected once or multiple times at* EA_{IS}*; the latter requires attribute* $fre \in \mathbb{N}$ *determining how often* PF_{IS} *shall be injected at* EA_{IS} *in parallel,*
- $InjTrigger$ *determines the point in time an injection is triggered. It is defined by a conditional predicate* $par\ \theta\ val$ *with* $par \in N \bigcup PP$*; further* $par \in N \Rightarrow NT(par) = DataObj, \theta \in \{``=``, `` \leq ``, \ldots\}, val \in Dom(par)$,
- $InjRank \in \mathbb{N}$ *is a number to create a ranking among injections specifications as they may match concurrently; all injection specifications for one particular extension area must expose different values,*
- $DR : InputData_{PF_{IS}} \rightarrow DO$ *is a set of mappings of input data objects* $InputData_{PF_{IS}}$ *of* PF_{IS} *to data objects* $DO \in N_{BP} \bigcup (N_{PF} \setminus N_{PF_{IS}})$ *of the base process* BP *or of other process fragments* $PF \setminus PF_{IS}$*,*
- $DW : OutputData_{PF_{IS}} \rightarrow DO$ *is a set of mappings of output data objects* $OutputData_{PF_{IS}}$ *of* PF_{IR} *to data objects* $DO \in N_{BP} \bigcup (N_{PF} \setminus N_{PF_{IS}})$ *of the base process* BP *or of other process fragments* $PF \setminus PF_{IS}$*.*

The injection trigger ($InjTriger$) enables the injection of a process fragment at an extension area as soon as a given process parameter or data object exposes a certain value (see Sect. 5.2 for details). Furthermore, the number of process fragments to be injected may be dynamically set based on the current contextual situation. Both concepts increase the flexibility provided to long-running and varying processes. The ranking ($InjRank$) of injection specifications becomes necessary as several contextual situations may occur concurrently and, hence, several injections (cf. Sect. 4.4) may be concurrently triggered. Through the ranking, especially, sequential injections of process fragments can be accomplished in a well-defined order. Sect. 5 presents details on context-aware process injection based on injection specifications.

5 The Process of Context-aware Process Injection

This section discusses the *process of context-aware process injection* to reveal the interplay and benefits of the introduced components and concepts. In particular, we show how to employ CaPI entities to properly inject process fragments at extension areas in given contextual situations. Thereby, we both discuss alternatives to specify CPFs at design time as well as the process of context-aware process injection at run time.

5.1 The Modeling of Context-aware Process Families

As a prerequisite, the base process of a context-aware process family must be defined first. Therefore, either a new process model needs to be created or an existing one is modified accordingly. Note that the resulting base process model solely contains the set of activities shared by all process variants, known at build time, and usually not being changed at run time. Drawing upon, the extension areas are then defined by selecting corresponding nodes in the base process.

Subsequently, the set of process parameters must be specified as the latter provides the basis for defining contextual situations and, finally, the injection specifications. In this context, a process modeler may demand a set of pre-defined process parameters that allow modeling interdependencies among process fragments. For example, the list of process fragments injected in the base process at run time may be made available through such a pre-defined process parameter. Note that this approach also allows for the incorporation of data objects, which belong to other process fragments, into the data mapping declared in an injection specification.

Based on the given process parameters and data objects, the set of contextual situations can be defined appropriately. The latter then enables a process modeler to finally define injection specifications. Altogether, three alternative modeling perspectives can be provided to a process modeler (cf. Fig. 6):

– *Situation-based perspective*: for every contextual situation, one may determine the process fragments to be injected at given extension areas.
– *Location-based perspective*: for each extension area, one may define the process fragments to be injected in a given contextual situation.
– *Artifact-based perspective:* one may stepwise take process fragments to define in which contextual situation they shall be injected at given extension areas.

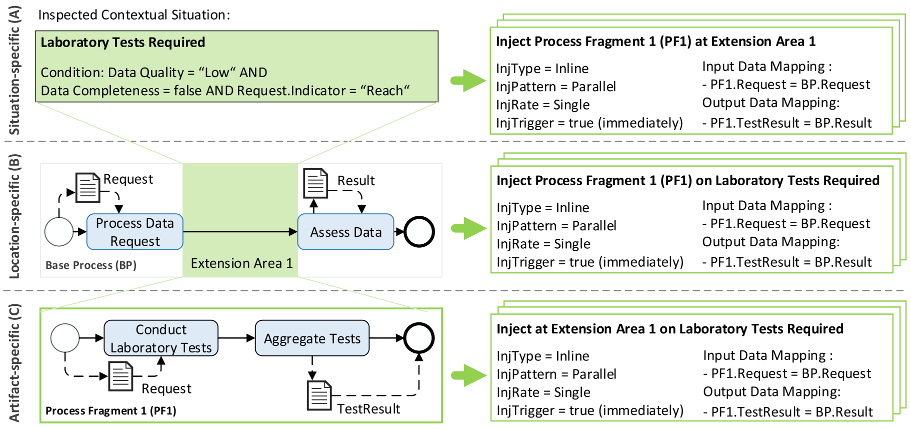

Situation-specific (A)
Inspected Contextual Situation:
Laboratory Tests Required
Condition: Data Quality = "Low" AND
Data Completeness = false AND Request.Indicator = "Reach"
Inject Process Fragment 1 (PF1) at Extension Area 1
InjType = Inline
InjPattern = Parallel
InjRate = Single
InjTrigger = true (immediately)
Input Data Mapping :
- PF1.Request = BP.Request
Output Data Mapping:
- PF1.TestResult = BP.Result
Location-specific (B)
Request
Process Data Request
Base Process (BP)
Extension Area 1
Result
Assess Data
Inject Process Fragment 1 (PF1) on Laboratory Tests Required
InjType = Inline
InjPattern = Parallel
InjRate = Single
InjTrigger = true (immediately)
Input Data Mapping :
- PF1.Request = BP.Request
Output Data Mapping:
- PF1.TestResult = BP.Result
Artifact-specific (C)
Conduct Laboratory Tests
Aggregate Tests
Request
TestResult
Process Fragment 1 (PF1)
Inject at Extension Area 1 on Laboratory Tests Required
InjType = Inline
InjPattern = Parallel
InjRate = Single
InjTrigger = true (immediately)
Input Data Mapping :
- PF1.Request = BP.Request
Output Data Mapping:
- PF1.TestResult = BP.Result

Fig. 6. Three Approaches for Modeling Injection Specification

As illustrated in Fig. 6, from each perspective the modeling still leads to the creation of injection specifications for the given CPF. However, a process modeler may use her favorite approach or even mix the approaches in relation to her personal preferences.

Since many activities and decisions in the control flow of the base process may be data-driven, the mapping of the injected data objects must be accomplished to successfully conduct CaPI. This very essential part for supporting process variants is consistently and easily achieved by selecting the data objects in both the base process model and the process fragments to create the required mapping (cf. Sect. 4.4). Note that this is a clear advantage of CaPI in comparison to many existing approaches (cf. Sec. 7) as the latter do not allow for (automatic) data mapping and, hence, process users are burdened with this issue at run time. As process fragments may be injected at different extension areas, one may want to link a data object to another data object of a process fragment injected earlier in the base process. Hence, a interdependency between such two process fragments must be created accordingly: process parameters providing meta information regarding the current execution trace (cf. Sect. 4.2) are leveraged to enhance the contextual situation for a process fragment. The latter can be easily automated as soon as one adds corresponding references to data objects of process fragments to be injected earlier. Finally, in case a process fragments is injected multiple times at an extension area, CaPI allows for referencing data objects of the injected fragments by adequate identification mechanisms.

5.2 The Execution of Context-Aware Process Families

As opposed to configuration approaches (cf. Sec. 7), CaPI enables the late configuration of processes at run time. The latter allows evaluating the contextual situations just in the moment a process adaptation is required. Therefore, a $CPF = (BP, EA, PP, CS, PF, IS)$ is deployed and executed in a process-aware information system (PaIS). After successful deployment, the base process instance $BPI = (BP, NS, \Pi)$ is continuously monitored by a dedicated CaPI application (cf. Sect. 6.1) continuously monitoring the BPI regarding reached extension areas and current contextual situations.

If an extension area $ea \in EA$ is reached and, especially, its first extension point refers to a node with scope `Pre` $(EP_s = (n_x, \texttt{Pre}), n_x \in N, BP = (N, E, NT, ET, EC))$, the determination of the contextual situations will be started as soon as the previous node will have been completed $(n_{x-1} \in N, NS(n_{x-1}) = \texttt{Completed})$. In turn, if the first extension point refers to a node with scope `Post` $(EP_s = (n_x, \texttt{Post}))$, the determination will be started as soon as n_x will have been completed $(NS(n_x) = \texttt{Completed})$. In case the extension area is surrounded by a loop in the BPI, the injection specification can be evaluated in the first iteration or in every iteration of the loop structure (depends on preferences and the support by th underlying PaIS). After successfully determining the set of contextual situations CS_{ea}, it becomes possible to derive the set of utilizable injection specifications IS_{ea} for finally adapt the BPI adequately.

For every process specification $is \in IS_{ea}$ with $is = (ea, cs_{is}, pf_{is}, InjType_{is}, InjPattern_{is}, InjRate_{is}, InjTrigger_{is}, InjRank_{is}, DR_{is}, DW_{is})$, the point of time the process injection shall be accomplished, must be regarded based on condition $InjTrigger_{is}$. If the latter is already met when reaching ea, pf_{is}

will be immediately injected according to the below-mentioned steps. Otherwise, the injection of pf_{is} will be postponed until $InjTrigger_{is}$ is fulfilled. In case the condition is never satisfied, pf_{is} could be injected at the very end of the control embraced by an extension area or not be injected at all (depends on preferences set initially). If pf_{is} shall be lately injected, the current states of the nodes embraced by ea must be taken into account: if there is only one running node $n_x \in N(i.e.NS(n_x) = \texttt{Running})$, pf_{is} will be injected directly after n_x. However, if there are several concurrently running nodes $n_k \in N, k = 1, \ldots, n(i.e.NS(n_k) = \texttt{Running})$, pf_{is} will be injected directly after the gateway finally merging the branches on which the nodes $n_1, \ldots, n_n$ are situated on. Finally, if several injection specifications are sharing the same contextual situations and injection trigger, the injection rank $InjRank_{is}$ is considered (cf. Sect. 4.4). After the consideration of $InjTrigger_{is}$ and $InjRank_{is}$, the following procedures are applied in general (cf. Fig. 7):

1. $InjPattern_{is} = \texttt{Parallel} \wedge InjRate_{is} = \texttt{Single}, \Rightarrow pf_{is}$ will be injected inline (or as a sub-process depending on $InjType_{is}$) and in parallel to the existing control flow,
2. $InjPattern_{is} = \texttt{Parallel} \wedge InjRate_{is} = (\texttt{Multiple}, \text{fre}), \Rightarrow pf_{is}$ will be injected inline (or as a sub-process) fre times with surrounding `ANDsplit` / `ANDjoin` gateways in parallel to existing control flow,
3. $InjPattern_{is} = \texttt{Sequential} \wedge InjRate_{is} = \texttt{Single} \Rightarrow, pf_{is}$ will be injected inline (or as a sub-process) into the existing control flow,
4. $InjPattern_{is} = \texttt{Sequential} \wedge InjRate_{is} = (\texttt{Multiple}, \text{fre}), \Rightarrow pf_{is}$ will be injected inline (or as a sub-process) fre times with surrounding `ANDsplit` / `ANDjoin` gateways into the existing control flow.

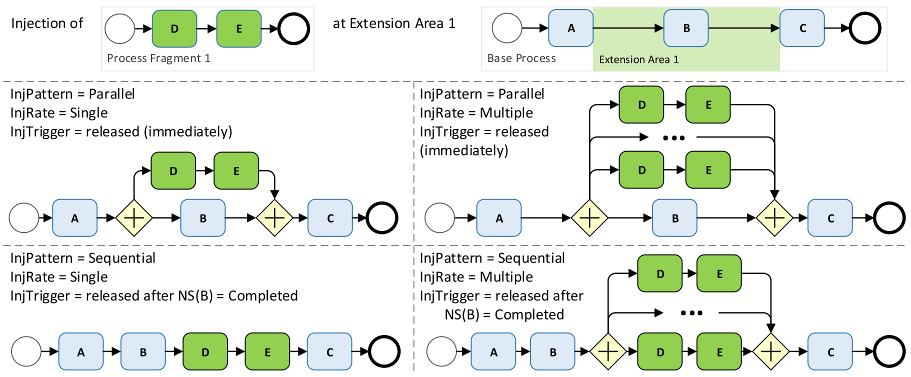


Fig. 7. Realization of Process Injection based on Injection Specifications

The detailed procedures to inject a process fragment pf_{is} are exemplarily discussed for the case $InjPattern_{is} = \texttt{Parallel} \wedge InjType_{is} = \texttt{Inline}$, assuming $InjTrigger_{is}$ has already been satisfied: first, start and end nodes $n^s_{pf_{si}}, n^e_{pf_{is}} \in N_{pf_{is}}$ of pf_{is} are removed. Then all remaining nodes $n^k_{pf_{is}} \in N_{pf_{is}}$

as well as one ANDsplit $n_{ANDsplit}$ and one ANDjoin gateway $n_{ANDjoin}$ are added to the nodes of the base process N_{BP}. Subsequently, six control edges are created: one edge connects n_p preceding the extension area with $n_{ANDsplit}$, two edges link $n_{ANDsplit}$ to $n^{s+1}_{pf_{is}}$, which succeeds the (removed) start node of pf_{is}, and n_{p+1}, which is the first node in the control flow embraced by ea. Subsequently, $n^{e-1}_{pf_{is}}$ (i.e, the last node of pf_{is}) and ea (i.e, last node of the control flow embraced by ea) are connected with $n_{ANDjoin}$. Finally, $n_{ANDjoin}$ is linked to n_s, which is the first node succeeding ea.

Finally, the correct data flow between injected nodes and existing nodes of the base process must be established. As this is automatically performed at run time, process participants are not burdened with this challenging task. We exemplarily present the data input mapping for the inline injection of a single fragment PF_{IS} (cf. Fig. 8): for every node $n_{pf_{is}}$ of pf_{is} with data edge $e_{di} = (di_{pf_{is}}, n_{PF_{IS}}) \in E, ET(e) = \texttt{DataEdge}$ from a input data object $di_{pf_{is}}, di_{pf_{is}} \in InputData_{pf_{is}}$, a new edge $e_{di\text{-}new} := (do_{BP}, n_{pf_{is}})$ is created based on mapping $dr = (di_{pf_{is}}, do_{BP}), dr \in DR$. e_{di} is deleted afterwards and if there are no further edges connecting $di_{pf_{is}}$ to nodes, $di_{pf_{is}}$ will be deleted as well.

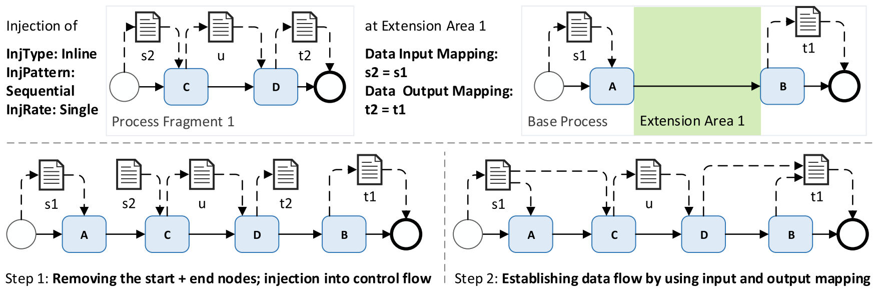


Fig. 8. Data Mapping Example for an Injected Process Fragment

6 Validation

As the CaPI approach explicitly addresses long-running processes showing high variability, which often take place in rather sensitive businesses, a mature and powerful implementation is required to conduct valuable empirical studies to successfully validate the concepts presented in this work. To prepare such studies, we developed a sophisticated proof-of-concept prototype whose details are presented in the following. Further, we conducted a first case study in the automotive and electronics industry to receive important feedback regarding both the approach in total as well as the proof-of-concept prototype in particular.

6.1 Proof-of-Concept Prototype

To establish a powerful implementation as a solid basis for future empirical studies, the CaPI proof-of-concept prototype is based on the conceptual architecture shown in Fig. 9. In particular, we realized the prototype using Aristaflow adaptive process management technology [6]. The latter allows modeling, deploying, and executing well-structured business processes. Further, it provides sophisticated and sound change operations to adapt running process instances at run time [16]. Hence, AristaFlow provides the basic execution platform required to conduct the sound injection of process fragments as well as the proper assignment of data objects for the injected activities at run time.

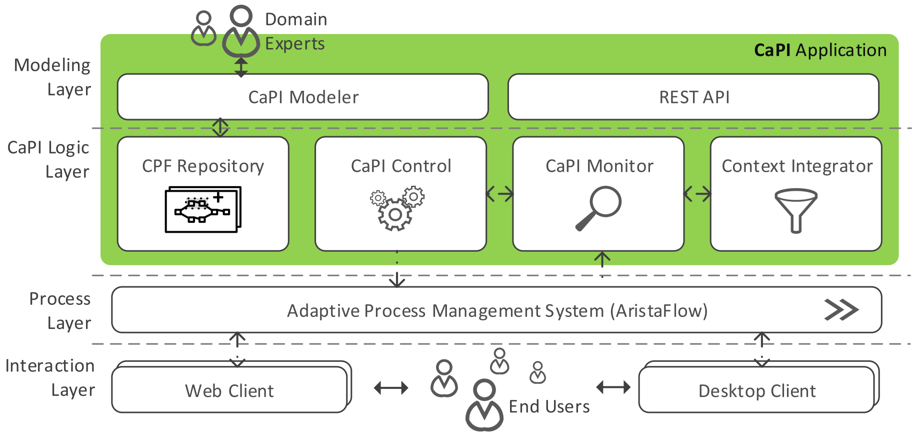
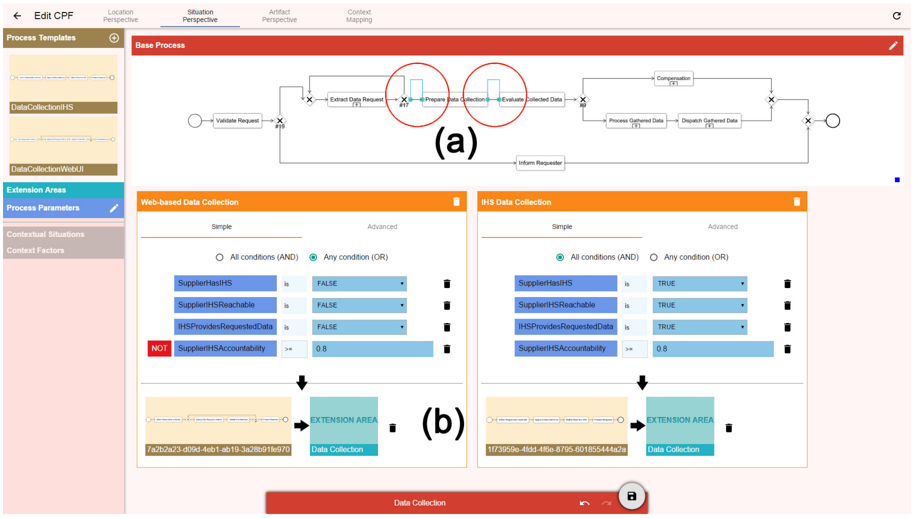


Fig. 9. Overview on the CaPI Architecture

Realized with Java Enterprise Edition 7, the CaPI application comprises a web-based sub-module enabling domain experts to conveniently model CPFs (*CaPI Modeler*) as well as sub-modules *CPF Repository*, *CaPI Monitor*, *CaPI Control*, and *Context Integrator* representing the CaPI core functions required at run time.

Through appropriate web-based user interfaces, a domain expert may first specify the mappings of the available context factors to process parameters and the one of the process parameters to contextual situations accordingly. Based on these preparations, she may create injection specifications by putting together the CaPI core components extension areas, contextual situations and process fragments via *drag and drop*. As proposed in Sect. 5.1, for this purpose, we implemented the different perspectives a domain expert may use to create an injection specification. Consequently, Fig. 10 exemplarily illustrates the situation-based perspectives showing a base process with two extension areas (see Marking (a)) for a data collection process regarding Reach Compliance of several suppliers (cf. Sect. 1). Both extension areas are needed to prepare and perform data collection activities for every involved supplier according to their capabilities (i.e. context factors). In particular, if a supplier hosts a well-reachable in-house system providing required data, the process fragment *"Perform Data Collection IHS"* is injected for every involved supplier at the second extension area.

Overall, the CPFs modeled by domain experts are managed in the CPF Repository. At run time, CaPI Control interprets the CPF specifications to detect the deployment of a CPF base process in AristaFlow and to continuously monitor the execution of the base process accordingly. Therefore, CaPI Control is registered as a dedicated service in AristaFlow to receive any status updates of activities as well as to actively acknowledge the start of every activity in the base process. Based on this approach, CaPI control detects when achieving an extension area, subsequently evaluates the valid contextual situations, and finally injects the specified process fragments on demand. For the example of Fig. 10, either *"Perform Data Collection IHS"* or *"Web-based Data Collection"* are injected at the second extension area according to the given contextual situations at run time (see Marking (b)).



Web-based Data Collection

All conditions (AND) [✓ Any condition (OR)]

	SupplierHasIHS	is	FALSE
	SupplierIHSReachable	is	FALSE
	IHSProvidesRequestedData	is	FALSE
NOT	SupplierIHSAccountability	>=	0.8

7a2b2a23-d09d-4eb1-ab19-3a28b91fe970 → Data Collection

IHS Data Collection

[✓ All conditions (AND)] Any condition (OR)

SupplierHasIHS	is	TRUE
SupplierIHSReachable	is	TRUE
IHSProvidesRequestedData	is	TRUE
SupplierIHSAccountability	>=	0.8

1f73959e-4fdd-4f6e-8795-601855444a2a → Data Collection

Fig. 10. Screenshots of the CaPI Situation-based Modeling Perspective

6.2 Case Study

After demonstrating the technical feasibility, we also conducted a case study based on data collection processes in the automotive and electronics industry (cf. Sect. 1) in the scope of the SustainHub[2] project. More precisely, we therefore modeled several data collection processes of an automotive manufacturer with its dynamic, data-driven injections of process fragments. Ensuing, we conducted qualitative interviews with project partners to receive their feedback. For the interviews, we presented both the approach and the existing functionality based on the modeled data collection process. Altogether, we received valuable, but of course limited feedback regarding *better modularity*, *increased confidentiality*, and *comprehensible monitoring*.

[2] SustainHub (Project No.283130) is a collaborative project within the 7th Framework Programme of the European Commission (Topic ENV.2011.3.1.9-1, Eco-innovation).

Regarding modularity, CaPI may reduce the complexity of long-running, varying process models to create more comprehensible and appropriate process models according to given contextual situations. The partners stated that the size and complexity of process models typically determine the rate of modeling errors. CaPI may provide a different way of modeling such complex and varying processes without these errors. However, the possibilities and ease-of-use regarding the modeling of contextual situations and injection specifications will mainly determine the effectiveness and efficiency of CaPI in comparison to the traditional approach of maintaining one large-sized process model. At design time, the systematic management of large process models also raises the problem of confidentiality. All possible decisions and activities including the linkage to roles, data, and other resources are accessible in total. According to the partners, modeling a process based on CaPI may provide possibilities to separate common activities and control flow from specific, confidential process fragments injected in contextual situations. Regarding confidentiality at run time, CaPI may provide only activities and control flow elements executed for monitoring purposes. Thereby, monitoring may be more comprehensible and descriptive in comparison to showing execution traces in large and complex process models.

7 Related Work

Classifying CaPI, we propose an implemented approach for the automated, context-aware extension of process instances at run time to cope with process variability and to increase process flexibility. Related work addresses the configuration of process models before deployment, the adaptation of process instances at run time, the late selection of sub-process, the late composition of services [3,17], and, in broader sense, aspect-oriented programming. In the following we discuss the commonalities and differences of related work in comparison to CaPI.

Approaches for process configuration, e.g., [9] or [7], aim at the modification of a reference process model to configure process model variants before process run time. Therefore, these approaches employ various transformations like adding process fragments, deleting activities, or changing control flow as well as properties of activities. However, these powerful transformations can be only applied, based on current information, before the process has been deployed. Instead, CaPI enables the injection of process fragments at run time. Further, CaPI considers context- and process-specific data at run time to support both process variability and process flexibility for long-running processes. Regarding automated adaptation of process instances at run time, rule-, case, and goal-based approaches may be taken into account [17]. Based on ECA (Event-Condition-Action), the rule-based approaches automatically detect exceptional situations and determine process instance adaptations required to handle these exceptions. Especially, AgentWork [13] is based on a temporal ECA rule model and enables automated structural adaptations of a running process instance (e.g., to add process fragments or to delete them) to cope with unplanned situations. However, CaPI entities allow for the specification of process variants instead of coping with unexpected failure events. Concretizing loosely specified processes, approaches for *late selection* typically rely on placeholder activities to integrate sub-processes in a base process at run time. While [1] suggests that the selection of the process fragment is primarily done by the process participants, [14] proposes an automatic, multi-staged approach to select sub-process at run time. Further, CaPI may be compared to process-based composition methods allowing for the late selection of service implementations [2,4,5]. These approaches share the abstract definition of a business process at design time. Each activity in the business process corresponds to a service specification and provides a placeholder for services matching the specification. Either upon invocation time or at run time, service implementations matching the specification are automatically selected from a registry based on QoS attributes or selection rules. By contrast, CaPI's extension areas in combination with injection specifications enable both the inline insertion of process fragments as well as the integration of process fragments as sub-processes. While placeholder activities are limited regarding the assignment of input and output data, the declaration of data mappings in the injection specifications enables the direct access to of activities in the process fragments to data objects in the base process.

Regarding related work in a broader sense, CaPI can be also well compared to *aspect-oriented programming* (AOP) [10]. AOP represents a programming paradigm for object-oriented programming and it targets high modularity by allowing and realizing the separation of system-level cross-cutting concerns from the actual key functionality. While AOP is also relying on injections at so-called *join points*, CaPI, by contrast, targets at the increased modularity of varying processes by separating activities, which are always performed, from activities and sub-processes performed in certain, pre-defined contextual situations.

8 Conclusion

In a nutshell, this work presents an approach for supporting long-running processes being subject to high variability by the context-aware and automated injection of process fragments at run time. Especially for long-running processes, the important configuration addressing process variety can hardly be performed solely at build time. However, existing approaches either focus on build-time configurations or allow for the late selection of process fragments based on placeholder activities. Consequently, the CaPI approach addresses this gap through providing context-aware configuration support at run-time based on the injection of process fragments. Finally, we further addressed the important data mapping for injected process fragments as well as we implemented a proof-of-concept prototype demonstrating the mentioned CaPI benefits.

In future research, we will conduct comprehensive experiments using the prototype to further examine the process of context-aware process injection. We further intend to enhance the CaPI modeler and to strengthen the context mapping by employing complex event processing.

Acknowledgement. This research was partially conducted within the SustainHub research project (Project No.283130) funded by 7th Framework Programme of the European Commission (Topic ENV.2011.3.1.9-1, Eco-innovation).

References

1. Adams, M., ter Hofstede, A.H.M., Edmond, D., van der Aalst, W.M.P.: Worklets: a service-oriented implementation of dynamic flexibility in workflows. In: Meersman, R., Tari, Z. (eds.) OTM 2006. LNCS, vol. 4275, pp. 291–308. Springer, Heidelberg (2006)
2. Aggarwal, R., Verma, K., Miller, J., Milnor, W.: Constraint driven web service composition in METEOR-S. In: Proc. SCC 2004, pp. 23–30 (2004)
3. Ayora, C., Torres, V., Weber, B., Reichert, M., Pelechano, V.: VIVACE: A framework for the systematic evaluation of variability support in process-aware information systems. Information and Software Technology **57**, 248–276 (2015)
4. Canfora, G., Di Penta, M., Esposito, R., Villani, M.L.: A framework for QoS-aware binding and re-binding of composite web services. J. Systems and Software **81**(10), 1754–1769 (2008)
5. Casati, F., Shan, M.C.: Dynamic and adaptive composition of e-services. Information Systems **26**(3), 143–163 (2001)
6. Dadam, P., Reichert, M.: The ADEPT Project: A Decade of Research and Development for Robust and Flexible Process Support - Challenges and Achievements. Computer Science - Research and Development **23**(2), 81–97 (2009)
7. Gottschalk, F., van der Aalst, W.M.P., Jansen-Vullers, M.H., La Rosa, M.: Configurable workflow models. Int. J. Coop. Inf. Sys. **17**(02), 177–221 (2008)
8. Grambow, G., Mundbrod, N., Steller, V., Reichert, M.: Challenges of applying adaptive processes to enable variability in sustainability data collection. In: SIMPDA 2013, pp. 74–88. CEUR Workshop Proceedings (2013)
9. Hallerbach, A., Bauer, T., Reichert, M.: Context-based configuration of process variants. In: Proc. TCoB 2008, pp. 31–40 (2008)
10. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
11. Lanz, A., Kreher, U., Reichert, M., Dadam, P.: Enabling process support for advanced applications with the aristaflow BPM suite. In: Proc. Business Process Management 2010 Demo Track. CEUR Workshop Proceedings (2010)
12. Müller, D., Reichert, M., Herbst, J.: A new paradigm for the enactment and dynamic adaptation of data-driven process structures. In: Bellahsène, Z., Léonard, M. (eds.) CAiSE 2008. LNCS, vol. 5074, pp. 48–63. Springer, Heidelberg (2008)
13. Müller, R., Greiner, U., Rahm, E.: AgentWork: a workflow system supporting rule-based workflow adaptation. Data & Knowledge Engineering **51**(2), 223–256 (2004)
14. Murguzur, A., De Carlos, X., Trujillo, S., Sagardui, G.: Context-aware staged configuration of process variants@runtime. In: Jarke, M., Mylopoulos, J., Quix, C., Rolland, C., Manolopoulos, Y., Mouratidis, H., Horkoff, J. (eds.) CAiSE 2014. LNCS, vol. 8484, pp. 241–255. Springer, Heidelberg (2014)

15. Peffers, K., Tuunanen, T., Rothenberger, M.A., Chatterjee, S.: A Design Science Research Methodology for Information Systems Research. J. Management Information Systems **24**(3), 45–77 (2007)
16. Reichert, M., Dadam, P.: ADEPTflex - Supporting Dynamic Changes of Workflows Without Losing Control. J. Intelligent Information Systems **10**(2), 93–129 (1998)
17. Reichert, M., Weber, B.: Enabling Flexibility in Process-Aware Information Systems: Challenges, methods, technologies. Springer, Heidelberg (2012)
18. Tiedeken, J., Reichert, M., Herbst, J.: On the Integration of Electrical/Electronic Product Data in the Automotive Domain. Datenbank Spektrum **13**(3), 189–199 (2013)
19. van der Aalst, W.M.P., ter Hofstede, A.H.M.: Verification of Workflow Task Structures: A Petri-net-based Approach. Inf. Sys. **25**(1), 43–69 (2000)