Pattern-driven Reuse in Architecture-centric Evolution for Service Software

Aakash Ahmad, Pooyan Jamshidi and Claus Pahl
School of Computing, Dublin City University, Dublin, Ireland
Lero - The Irish Software Engineering Research Centre, Limerick, Ireland

Keywords: Software Evolution, Change Patterns, Architecture Model Evolution.

Abstract: Service-based architectures implement business processes as technical software services to develop enterprise software. As a consequence of frequent business and technical change cycles, the architect requires a reuse-centered approach to systematically accommodate recurring changes in existing software. Our *'Pat-Evol'* project aims at supporting pattern-driven reuse in architecture-centric evolution for service software. We propose architecture change mining as a complementary phase to a systematic architecture change execution. Therefore, we investigate the 'history' of sequential changes - exploiting change logs - to discover patterns of change that occur during evolution. To foster reuse, a pattern catalogue maintains an updated collection with once-off specification for identified pattern instances. This allows us to exploit change pattern as a generic, first class abstractions (that can be operationalised and parameterised) to support reuse in architecture-centric software evolution. The notion of 'build-once, use-often' empowers the role of an architect to model and execute generic and potentially reusable solution to recurring architecture evolution problems.

1 INTRODUCTION

Software architecture represents the global system structure for designing, evolving and reasoning about configurations of computational components and their interconnection at higher abstraction levels. Service Oriented Architecture (SOA) represents a business-centric, architectural approach to model business process as technical software services in enterprise software. Once deployed, a continuous change in business and technical requirements lead towards frequent maintenance and evolution in service-driven software (Lewis and Smith, 2008). Although architecture-centric maintenance and evolution proved successful in adjusting software structure and behavior at higher abstractions, a systematic investigation (Breivold et al., 2012) suggests a lack of concrete efforts in supporting a reuse-centered approach to architecture evolution. From an architect's perspective (Garlan et al., 2009), this is particularly limiting as the existing solutions fall short of capitalising on the 'build-once, use-often philosophy'.

The potential for change reuse is also evident in the research taxonomy for SOA maintenance and evolution (Lewis and Smith, 2008), urging development of processes, frameworks and patterns to foster reusable evolution in service-driven software. Within the *'Pat-Evol'* project (Ahmad and Pahl, 2012a) we aim at supporting pattern-driven reuse in architecture-centric evolution for service software. We rely on architecture change mining (analysing implicit evolution-centric knowledge) as a complementary phase to guide architectural change execution. In (Ahmad et al., 2012) we investigated the history of sequential architectural changes - exploiting change logs - to empirically identify patterns of change that occur over-time during evolution. This allows us to hypothesise that *the application of change patterns to architectural transformation supports potential reuse in architecture-centric software evolution*. Change patterns as a generic solution can be i) identified as recurrent, ii) specified once and iii) instantiated multiple times to support reuse in architectural change.

Although a recent emergence of evolution styles (Garlan et al., 2009) and more specifically the 'Evolution Shelf' (Goaer et al., 2008) promote reuse in (more conventional) component-based architecture evolution. However, these solutions fall short of addressing frequent demand-driven process-centric changes (Weber et al., 2007) that are central to maintenance and evolution of SOAs. This motivates the needs to systematically investigate architecture

Ahmad A., Jamshidi P. and Pahl C..
Pattern-driven Reuse in Architecture-centric Evolution for Service Software.
DOI: 10.5220/0004128602790284
In *Proceedings of the 7th International Conference on Software Paradigm Trends* (ICSOFT-2012), pages 279-284
ISBN: 978-989-8565-19-8

change representation that goes beyond frequent addition or removal of individual components and connectors to operationalise recurrent process-based architectural changes. This is significant in relieving an architect of routine evolution tasks by fostering their reuse to support a systematic change execution whenever needs for architectural evolution arises.

In the *'Pat-Evol'* project, the overall research contribution in terms of an evolution application framework is depicted in Figure 1. In this paper, we specifically focus on evolution specification that is guided by architecture change patterns as summarised below.

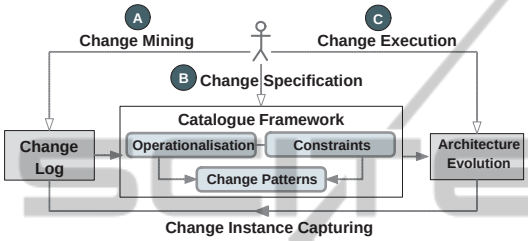

Figure 1: An overview of solution framework.

A. *We analysed the history of sequential architectural changes as 'post-mortem' analysis to continuously identify recurring operationalisation and patterns from change logs (Ahmad et al., 2012).*

B. *We support pattern specification and retrieval with a repository infrastructure comprising of a continuously validated and updated collection of change patterns (Ahmad and Pahl, 2012b).*

C. *In this paper, we focus on evolution application in terms of pattern-driven reuse during change execution to support the notion of 'evolution off-the-shelf' in service-based architectures.*

This paper is organised as follows. A formal specification of change patterns and its properties are presented in Section 2. We discuss pattern-driven architecture evolution in Section 3. Section 4 outlines related research to justify overall contribution that is followed by conclusions and outlook in Section 5.

2 CHANGE PATTERN

In (Ahmad et al., 2012), we focused on exploiting architecture change logs - analysing sequential architectural changes - to empirically identify patterns of change that occur over time. This allows us to define change pattern as *a generic, first class abstraction (that can be operationalised and parameterised) to support potential reuse in architectural change execution*. In this section, we i) formalise change pattern in terms of a meta-model of its constituent elements in Figure 2 and ii) outline pattern properties formulating the foundation for pattern-driven change execution.

2.1 Modeling Pattern-based Evolution

We model pattern-based evolution as 4-tuple $PatEvol =< SArch, OPR, CNS, PAT >$ with element inter-relationships in Figure 2 as explained below.

- Service Architecture (SArch): refers to architecture elements to which a change pattern can be applied. We use attributed typed graphs (ATG) (Ehrig et al., 2004) that provide formal syntax and semantics with node and edge attribution to model typed instances of architectural elements. The architectural model is consistent with the Service Component Architecture with configurations (CFG) of a set service components (CMP) as computational entities linked through connectors (CON). Following SOA principles (Lewis and Smith, 2008), modeling is restricted to graph-based representation of service architectures, only supporting composition or association type dependencies in service composites (Pahl, 2002; Pahl and Zhu, 2006). Thus, the consistency of pattern-based change and structural integrity of architecture elements beyond this architecture model is undefined.

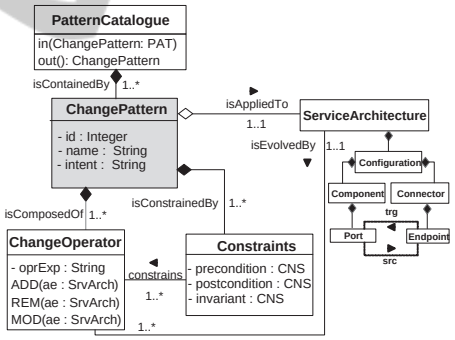

Figure 2: Structural model for pattern-based evolution.

- Change Operators (OPR): represents change operationalisation that is fundamental to architectural evolution. Our analysis of the log goes beyond basic change types in (Buckley et al., 2005) to specify a set of atomic and composite operations enabling structural evolution by adding (ADD), removing (REM) and modifying (MOD) architecture elements. The benefit with graph-based modeling lies in abstracting change operationalisation as declarative graph transformation rules (Corradini et al., 1996) specified as:

Atomic Change Operations: enable fundamental architectural changes in terms of adding, removing or modifying the component operation (OPT), component port (POR), connector binding (BIN), connector

endpoint (EPT) and configuration interface (INF).

Composite Change Operations: are a set of atomic change operations, combined to enable composite architectural changes. These enable adding, removing or modifying the components (CMP), connectors (CON) and configurations (CFG) with a sequential composition of architectural changes.

- Constraints (CNS): refer to a set of pattern-specific constraints in terms of pre-conditions (PRE) and post-conditions (POST) to ensure consistency of pattern-based changes. In addition, the invariants (INV) ensure structural integrity of individual architecture elements during change execution.

- Change Patterns (PAT): represents a recurring, constrained composition of change operationalisation on architecture elements. It defines a first class abstraction that can be operationalised and parameterised to support potentially reusable architectural change execution as: $PAT_{<id,\ name>} : PRE(ae_m \in AE) \xrightarrow{INV(OPR_n(ae_m \in AE))} POST(ae'_m \in AE)$.

A **pattern catalogue (CAT)** refers to a template-based repository infrastructure that facilitates an automated storage (in: once-off specification) and retrieval (out: multiple instantiation) of change patterns.

2.2 Properties of Change Pattern

Based on the meta-model, **Pattern Identification** is an empirical investigation of the history of architectural changes to identify recurring sequences of change. In (Ahmad et al., 2012), we use graphs to analyse change representation along with an automated identification of patterns from architecture change logs. **Pattern Specification** provides a consistent (once-off) specification for the collection of identified pattern instances. In (Ahmad and Pahl, 2012b), we use an XML-based pattern template to document generic change patterns that can be retrieved whenever needs for pattern usage arises. Finally, **Pattern Instantiation** allows instantiation of appropriate pattern instances from abstract specifications to promote the 'build-once, use-often' approach. Now, we focus on utilising graph transformation guided by architectural change patterns to support reusable evolution.

3 PATTERN-BASED EVOLUTION IN SERVICE ARCHITECTURE

Based on the operationalisation of architectural changes in the log, we observed recurring sequences of change that corresponded to evolution needs and their underlying tasks that occur frequently over time. Furthermore, if design or architectural patterns can prove to be successful in supporting reuse for architectural construction, we believe change patterns have the potential to facilitate reuse during its evolution. In particular, fostering the reuse of routine evolution tasks is an attempt towards facilitating architects to follow a systematic, reuse-centered approach to architecture-based change execution as highlighted in (Garlan et al., 2009). It is vital to mention that architectural evolution in the context of services-driven software is particularly inclined towards accommodating process-centric changes (Weber et al., 2007) (i.e., integration, replacement, decomposition of elements), unlike frequent addition or removal of individual components and connectors.

3.1 Evolution Use Case Scenario

We use a partial architectural view of an Electronic Bill Presentment and Payment (EBPP) as a case study. For architectural modeling, we use ATGs (Ehrig et al., 2004) to model typed instances of architectural elements. We use the Graph Modeling Language (.GML) as an XML-based representation of architectural instances. A benefit of graph-based modeling is support for architectural evolution by means of graph transformations (Corradini et al., 1996), (Graaf, 2007). More specifically, during execution change operationalisation is abstracted as declarative graph transformation rules (in our case XML transformations using XSLT). For space reasons we abstract the underlying graph-based formalism with an overview of pattern-driven transformation to evolve architectural model.

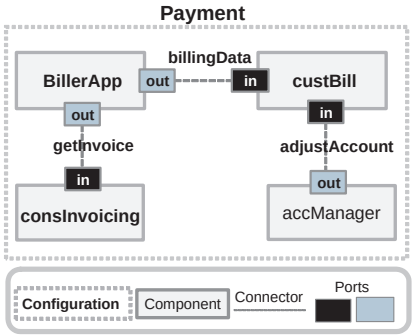

Figure 3: Partial architectural view for EBPP.

In the existing scope of case study (Figure 3), the company charges customer with full payment of customer bills in advance to deliver requested services. Now, the company plans to facilitate existing customers with either weekly or monthly bill payment. Therefore, in the given architectural (Figure 3) context, *a new component billType must be included and*

integrated as a mediator among the existing components BillerApp and custBill. It facilitates billing type options (monthly or weekly billing) to the customers.

3.2 Pattern-based Change Execution

In order to achieve pattern-driven architectural change execution, we follow a three step process as presented in Table 1 and explained below.

Table 1: Declarative specification for change intent.

$Integrate(G_S, [CNS], < billType, BillerApp, custBill >)$
PRE (preConditions)
$\exists(BillerApp, custBill \in CMP) \subseteq G_S$
$\exists billingData(BillerApp, custBill) \in CON \subseteq G_S$
POST (postConditions)
$\exists(BillerApp, custBill, billType \in CMP) \subseteq G_S$
$\exists billTypeData(BillerApp, billType) \in CON \subseteq G_S$
$\exists getType(billType, custBill) \in CON \subseteq G_S$
$<$ **PAT** $>$ **retrievePattern**(**PRE**, **POST**)
$\forall pat_i \in PAT \exists pat_i.PRE \wedge pat_i.DEF \wedge pat_i.POST$
$(pat_i.PRE \equiv PRE \wedge pat_i.POST \equiv POST) \in CAT$
$return(pat_i.DEF)$

1. *Change Specification:* A declarative specification allows architect to represent syntactical context of architectural change that contains the i) source architecture graph (G_S), ii) pre- and post-conditions ($[CNS]$) and iii) architecture elements (AE) that need to be added, removed or modified such that $AE \in G_S$. For example, in Table 1 the declaration Integrate() specifies that the new component billType is added in G_S and integrated as a mediator among the existing components BillerApp and custBill using the predefined constraints.

2. *Change Conditions:* consists of preconditions (PRE) and postconditions (POST) that must be satisfied to preserve the structural integrity of the overall architecture and individual elements during change execution. In addition, the catalogue (CAT) is queried based on these conditions to retrieve a list of patterns (PAT) in a given context.

   The **preconditions** represent the context of architectural elements before change execution. For example in Figure 4a, the sub-graph S specifies the exact sub-architecture billingData(BillerApp, custBill) that is subject to change in the original architecture G_S, PRE in Table 1. In order to apply changes, we must find an exact structural match m_1 of S in G_S, such that $m_S : S \rightarrow G_S$ as Figure 4a.

   The **postconditions** specify the context of evolved architectural elements as a result of the change execution. After applying changes on specified elements the overall architectural structure must be preserved. In order to include the modified architecture elements T in the target architecture G_T an exact structural match m_T of T in G_T must exist such that $m_T : T \rightarrow G_T$ as Figure 4c.

   The **invariants** represent architectural structure that is never changed during evolution. This is represented as Figure 4b, the invariant graph G_I as $m_I : I \rightarrow G_I$ with Double-Push-Out (DPO) graph transformations (Corradini et al., 1996).

3. *Pattern Retrieval:* Once an exact instance of S in G_S is identified, the pattern catalogue is queried with pre-conditions and post-conditions to retrieve the appropriate pattern that provides the potential reuse of change operationalisation to enable architectural evolution. Table 1 outlines the generic syntax for querying the catalogue based on Xpath query to retrieve pattern specification. The query matches the specified change pre-conditions and post-conditions to retrieve the pattern definition from catalogue. Figure 4 illustrates the retrieved instance of Linear Inclusion pattern that aims at *'integration of a mediator among two or more directly connected service components'*. In addition, **pattern instantiation** involves labeling of generic elements in specification with labels of concrete architecture elements presented in change specification. For example, in Figure 4a the connector instance billingData that is missing in the change post-conditions is removed from the G_S. The newly added instances of component billType and connector billTypeData, getType are the candidates for addition into G_S to obtain G_T in Figure 4c.

In example above, we support the fundamental hypothesis that: *if an architectural evolution problem can be specified declaratively, its solution is executed in an automated way in terms of instantiating change operationalisation that exist in the catalogue*. Keeping in view some technical clarifications we provide a brief overview of the change execution that is facilitated using the DPO construction (Corradini et al., 1996). In Figure 4 the order of change operations is insignificant and the sequence is presented as it appeared in the given pattern instance.

- **Deletion:** In Figure 4b, $S \setminus I$ describes the architecture elements which are to be deleted from G_S, For example, the connector billingData is removed from the BillerApp and custBill. The invariant graph G_I is obtained from the source graph G_S for elements which are a pre-image in S but not in I - the deletion operation - as $Rem_{<billingData \in CON>}$.
- **Addition:** In Figure 4c, $R \setminus I$ described the part which needs to be added in G_S to obtain G_T during change execution. For example, in Fig-

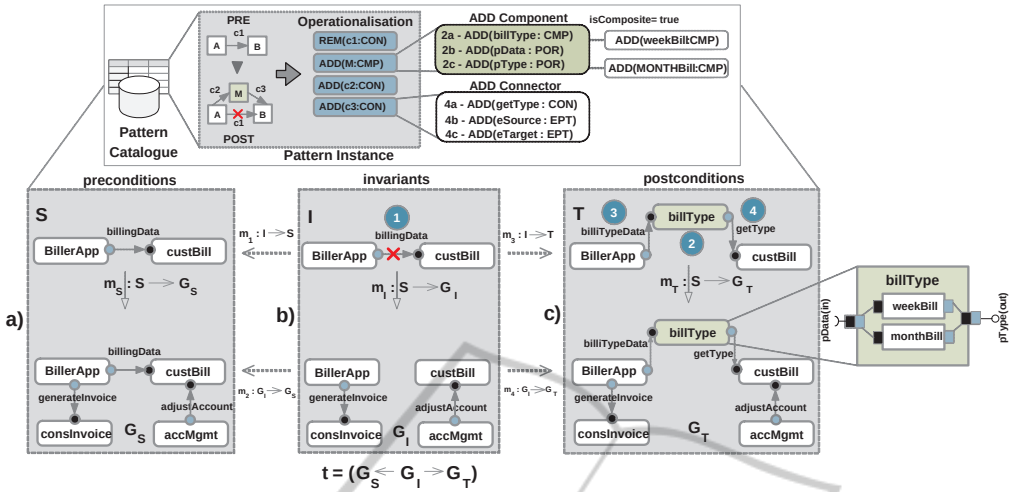

Figure 4: Pattern-driven architecture change execution.

ure 4c the component PaymentType is added with connector selectType and custPay in the architecture. $Add_{<billType \in CMP>}$; $Add_{<billTypeData \in CON>}$; $Add_{<getType \in CON>}$.

3.3 Evaluation Plan

A performant solution still requires a rigorous evaluation in terms of different architecture evolution scenarios. This is particularly beneficial to analyse the adequacy of the proposed solution in facilitating a reuse-centered approach to architecture evolution.

Scenario-based Evaluation: In order to avoid scalability issues, preliminary evaluation of our work is based on experimenting with evolutionary scenarios in two service-based architectures. These include an Electronic Bill Presentment and Payment and an On-line Tour Reservation System. The objectives of initial evaluation are to determine the adequacy of proposed solution in terms of utilising recurring operationlisations that exist in the catalogue.

Possible Limitations - An interesting identification is the emergence of change anti-patterns resulting from counter-productive pattern-based evolution. This leads us to believe that change patterns do not necessarily support an optimal solution; instead they promote an alternative and potentially reusable operationalisation for architecture evolution. However, identification of possible anti-patterns and their resolution (architectural refinement) is vital to achieve evolved architecture with desired specifications

Prototype-based Validation: Currently, we are in the process of developing a prototypical framework to fully automate and validate pattern-driven architecture evolution. The tool and the generated data shall form the basis for the practitioners (software architects) for a survey and usability based analysis to evaluate the adequacy and applicability of the proposed solution in a practical context.

4 RELATED WORK

During software design and evolution phase, an inherent limitation with the architectural description languages (ADLs) or meta-models lies with their inability to capitalise on an explicit change support within existing architecture. Although xADL (Dashofy et al., 2002) utilises the configuration management concepts - utilising version graphs - to capture the evolution of individual elements in architectural description, it lacks the granularity of change representation from one version to another. This is particularly limiting to i) model and analyse the operational representation of changes and ii) fostering their (potential) reuse to guide architectural change execution whenever needs for evolution arises. In contrast to the “evolution shelf” (Goaer et al., 2008), we are specifically concerned about frequent demand-driven, process-centric changes that are central to maintenance and evolution in service-driven architectures (Weber et al., 2007), (Lewis and Smith, 2008).

In the context of model driven engineering, the research in (Graaf, 2007) specifically focuses on architectural abstractions by exploiting model transformations to enable architecture model evolution. Our research is fundamentally different and supports a reuse-centered approach to achieve evolution in (platform independent) architectural model. However, the proposed solution is closely aligned to the work in (Kim and Carrington, 2006) and (Lara and Guerra, 2008) that allows users to define patterns using a pat-

tern modeling notations. These support pattern-based model evolution where patterns are used as transformation rules to support model evolution. We primarily focused on utilising experimentally identified patterns that facilitate a first class abstraction to reusable change execution. This enables fine granular change operationalisation that can be parameterised to guide potentially reusable evolution in architecture-centric models. It is vital to mention about a catalog of process change patterns (Weber et al., 2007) that guides changes in process-aware information systems. In contrast to the process aspects of a software, we exclusively focus on change operationalisation at architectural abstraction levels accommodating patterns to guide structural evolution.

5 CONCLUSIONS AND OUTLOOK

We propose to leverage architectural change mining - identifying patterns from change logs - to support potential reuse during architecture evolution. We demonstrate that if an architectural evolution problem can be specified declaratively, pattern-driven evolution could relieve an architect from the underlying operational concerns for executing routine evolution tasks that exist in catalogue.

In future, we are primarily concerned about explicitly addressing the granularity of change execution that goes beyond generic specifications of identified patterns. We focus on structural modifications to the internal architecture of composite components guided by change patterns that is currently lacking in our solution. Additional case studies in (Gruhn et al., 1996) and (Javed et al., 2009) shall allow a more rigorous evaluation in terms of a solution that facilitates the notion of 'build-once, use-often philosophy' to evolve architectural models.

REFERENCES

Ahmad, A., Jamshidi, P., and Pahl, C. (2012). Graph-based Pattern Identification from Architecture Change Logs. In *Intl Workshop On System/Software Architectures*.

Ahmad, A. and Pahl, C. (2012a). Pat-Evol: Pattern-Driven Reuse in Architecture-Based Evolution for Service Software. In *ERCIM News 88*.

Ahmad, A. and Pahl, C. (Retrieved April 1, 2012b). Pattern Catalogue: Towards an Automated Storage and Retrieval of Architecture Change Patterns (Technical Report). *http://www.computing.dcu.ie/ aaakash/PatternCatalogue.pdf*.

Breivold, H. P., Crnkovic, I., and Larsson, M. (2012). A Systematic Review of Software Architecture Evolution Research. *Inf and Softw Techn*, 54(1):16–40.

Buckley, J., Mens, T., Zenger, M., Rashid, A., and Kniesel, G. (2005). Towards a Taxonomy of Software Change. *Journal of Software Maintenance and Evolution*, 17:309–332.

Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., and Löwe, M. (1996). Algebraic Approaches to Graph Transformation, Part I: Basic Concepts and Double Pushout Approach. In *Handbook of Graph Grammars and Computing Graph Transformation, Volume 1: Foundations*, pages 163–245.

Dashofy, E., v.d. Hoek, A., and Taylor, R. N. (2002). An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. In *24th International Conference on Software Engineering.*

Ehrig, H., Prange, U., and Taentzer, G. (2004). Fundamental Theory for Typed Attributed Graph Transformation. In *Graph Transformations*, pages 161–177.

Garlan, D., Barnes, J., Schmerl, B., and Celiku, O. (2009). Evolution Styles: Foundations and Tool Support for Software Architecture Evolution. In *Proc. Joint Working IEEE/IFIP Conference on Software Architecture*.

Goaer, O. L., Tamzalit, D., Oussalah, M., and Seriai, A. D. (2008). Evolution Shelf: Reusing Evolution Expertise within Component-Based Software Architectures. In *IEEE Intl Computer Software and Applications Conf*.

Graaf, B. (2007). Model-Driven Evolution of Software Architectures. *Ph.D Dissertation: T.U. Delft.*

Gruhn, V., Pahl, C., and Wever, M. (1996). Data Model Evolution as Basis of Business Process Management. In *14th International Conference on Object-Oriented and Entity Relationship Modelling O-O ER95*. Springer-Verlag (LNCS Series).

Javed, M., Abgaz, Y.M., Pahl, C. (2009). A pattern-based framework of change operators for ontology evolution. In: On the Move to Meaningful Internet Systems: OTM Workshops, *LNCS* 5872, pp. 544–553. Springer.

Kim, S. K. and Carrington, D. (2006). A Pattern based Model Evolution Approach. In *13th Asia Pacific Software Engineering Conference*.

Lara, J. and Guerra, E. (2008). Pattern-Based Model-to-Model Transformation. In *4th International Conference on Graph Transformations*.

Lewis, G. and Smith, D. (2008). Service-Oriented Architecture and its Implications for Software Maintenance and Evolution. In *Frontiers of Software Maintenance*.

Pahl, C. (2002). A Formal Composition and Interaction Model for a Web Component Platform. ICALP'2002 Workshop on Formal Methods and Component Interaction. Elsevier ENTCS.

Pahl, C. and Zhu, Y. (2006). A Semantical Framework for the Orchestration and Choreography of Web Services. International Workshop on Web Languages and Formal Methods WLFM'05. Elsevier ENTCS.

Weber, B., Rinderle, S., and Reichert, M. (2007). Change Patterns and Change Support Features in Process-Aware Information Systems. In *19th Intl. Conference on Advanced Information Systems Engineering*.