Supporting Structural Consistency Checking in Adaptive Case Management

Christoph Czepa[1], Huy Tran[1], Uwe Zdun[1], Stefanie Rinderle-Ma[1], Thanh Tran Thi Kim[2], Erhard Weiss[2], and Christoph Ruhsam[2]

[1] Faculty of Computer Science, University of Vienna, Austria
{christoph.czepa,huy.tran,uwe.zdun,stefanie.rinderle-ma}@univie.ac.at
[2] Isis Papyrus Europe AG, Maria Enzersdorf, Austria
{thanh.tran,erhard.weiss,christoph.ruhsam}@isis-papyrus.com

Abstract. Adaptive Case Management (ACM) enables knowledge workers to collaboratively handle unforeseen circumstances by making ad hoc changes of case instances at runtime. Therefore, it is crucial to ensure that various structural elements of an ACM case, such as goals, subprocesses and so on, remain consistent over time. To the best of our knowledge, no studies in the literature provide adequate support for structural consistency checking of ACM. In this paper, we introduce a formal categorization of ACM's structural features and potential inconsistencies. Based on this categorization, we develop a novel approach for structural consistency checking of ACM cases. Our approach, based on model checking and graph algorithms, can detect a wide range of inconsistencies of ACM's structural elements. The evaluation of our approach shows reasonable performance and scalability.

1 Introduction

Adaptive Case Management (ACM) is part of an emerging trend in the field of business process management that is aiming at supporting highly flexible, knowledge-intensive software systems [13]. A high degree of flexibility often comes at the cost of potential errors or inconsistencies, subsequently hampering the executability of the processes. In the context of ACM, new sources of inconsistencies occur when compared to flexible process management technology (cf. e.g., [11]). For example, a new goal could be in conflict with an existing goal or a new dependency among two tasks could make no sense because of contradicting pre- and postconditions of the tasks. Structural consistency checking focuses on revealing such internal inconsistencies and contradictions inside the structure of a case.

In this paper, we propose a novel approach for supporting structural consistency checking of ACM systems. First, we study existing ACM systems and relevant industrial standards and introduce a formal categorization of ACM's structural features and potential inconsistencies. Based on this categorization, we developed checking techniques that can discover a wide range of inconsistencies of ACM cases. Our prototype shows reasonable performance and scalability, so the proposed approach is appropriate for both runtime and design time checking.

2 ACM Structural Concepts and Possible Inconsistencies

In this section, we identify ACM's structural concepts and which inconsistencies can possibly occur in these structures.

2.1 Formal Definition of ACM Case Models

Our identification of ACM's structural concepts is based on a generalization of the commercial ACM solution ISIS Papyrus[3], the CMMN standard[4], and the existing ACM literature [6, 9, 14]. Based on this study we have derived a formal definition of a case model.

Definition 1. *A case model $\mathcal{M}$ is a tuple $(\mathcal{T}, \mathcal{G}, \mathcal{S}, \mathcal{E}, \mathcal{X}, \mathcal{C}, \mathcal{D}, \zeta_{\mathcal{E}}, \zeta_{\mathcal{X}}, \eta, \mathcal{T}_{\mathcal{F}}, \mathcal{F}, \phi, \mathcal{J}, \delta, s_m)$ where*

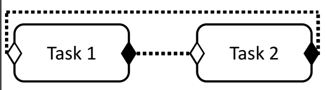
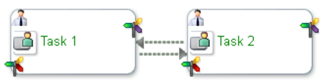
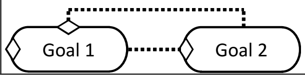
- *$\mathcal{T}$ is a set of tasks, $\mathcal{G}$ is a set of goals, $\mathcal{S}$ is a set of stages, $\mathcal{E}$ is a set of entry criteria, $\mathcal{X}$ is a set of exit criteria, $\mathcal{C} = \mathcal{E} \cup \mathcal{X}$ is a set of criteria,*
- *$\mathcal{D} = \mathcal{D}_{\mathcal{X}\mathcal{E}} \cup \mathcal{D}_{\mathcal{X}} \cup \mathcal{D}_{\mathcal{E}} \cup \mathcal{D}_{\mathcal{S}_{\mathcal{X}}} \cup \mathcal{D}_{\mathcal{S}_{\mathcal{E}}} \cup \mathcal{D}_{\mathcal{N}}$ is a set of dependencies, where*
  - *$\mathcal{D}_{\mathcal{X}\mathcal{E}} \subsetneq \mathcal{X} \times \mathcal{E}$ is a set of dependencies from exit to entry criteria,*
  - *$\mathcal{D}_{\mathcal{X}} \subsetneq \mathcal{X} \times (\mathcal{T} \cup \mathcal{S})$ is a set of dependencies from exit criteria to tasks, and stages,*
  - *$\mathcal{D}_{\mathcal{E}} \subsetneq (\mathcal{T} \cup \mathcal{G} \cup \mathcal{S}) \times \mathcal{E}$ is a set of dependencies from tasks, goals, and stages to entry criteria,*
  - *$\mathcal{D}_{\mathcal{S}_{\mathcal{X}}} \subsetneq (\mathcal{T} \cup \mathcal{G} \cup \mathcal{S} \cup \mathcal{X}) \times \mathcal{X}$ is a set of dependencies from stage internal tasks, stages, goals, and exit criteria to exit criteria of the stage.*
  - *$\mathcal{D}_{\mathcal{S}_{\mathcal{E}}} \subsetneq \mathcal{E} \times (\mathcal{T} \cup \mathcal{S} \cup \mathcal{E})$ is a set of dependencies from entry criteria of a stage to stage internal tasks, stages, and entry criteria.*
  - *$\mathcal{D}_{\mathcal{N}} \subsetneq (\mathcal{T} \cup \mathcal{G} \cup \mathcal{S}) \times (\mathcal{T} \cup \mathcal{S})$ is a set of dependencies from tasks, goals and stages to tasks and stages.*
- *$\zeta_{\mathcal{E}} : \mathcal{E} \mapsto \mathcal{T} \cup \mathcal{G} \cup \mathcal{S}$ is a total non-injective function which maps an entry criterion to a task, goal, or stage,*
- *$\zeta_{\mathcal{X}} : \mathcal{X} \mapsto \mathcal{T} \cup \mathcal{S}$ is a total non-injective function which maps an exit criterion to a task or stage,*
- *$\eta : \mathcal{G} \mapsto \mathcal{G}$ is a partial non-injective function which maps a goal to a parent goal*
- *$\mathcal{T}_{\mathcal{F}} \subsetneq \mathcal{T}$ is a set of process tasks, $\mathcal{F}$ is a set of subprocesses,*
- *$\phi : \mathcal{T}_{\mathcal{F}} \mapsto \mathcal{F}$ is a total function which maps a process task to a subprocess,*
- *$\mathcal{J} = \mathcal{C} \cup \bigcup_{p \in \mathcal{F}} p.\mathcal{B}$ is the joint set of criteria and conditions of all subprocesses,*
- *$\delta : \mathcal{T} \cup \mathcal{G} \cup \mathcal{S} \mapsto \mathcal{S}$ is a partial non-injective function which maps a task, goal, or stage to a parent stage, and*
- *$s_m \in \mathcal{S}$ is the main stage of the case (equivalent to a Case element in CMMN).*

2.2 Possible Inconsistencies in ACM's Structural Concepts

Dependencies among elements in ACM can be arranged to form loops. Table 1 shows two kinds of loop-related inconsistencies in ACM. If the elements in a loop formed by

[3] `http://www.isis-papyrus.com`
[4] `http://www.omg.org/spec/CMMN/1.0/PDF`

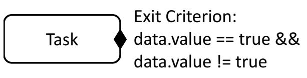
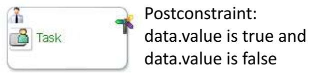
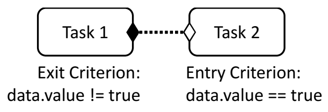
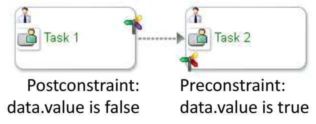
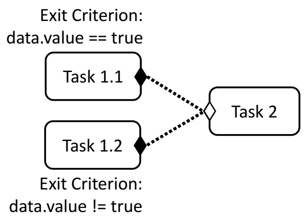
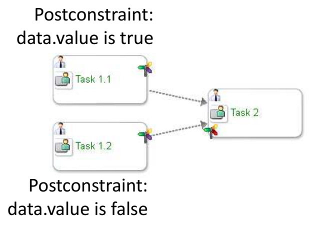
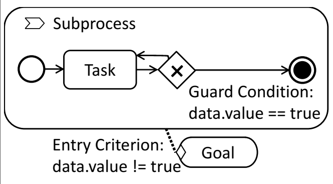
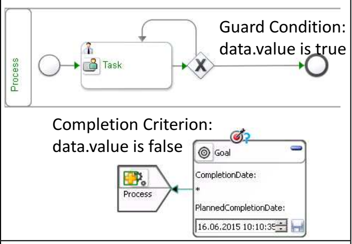
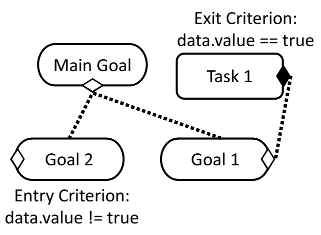
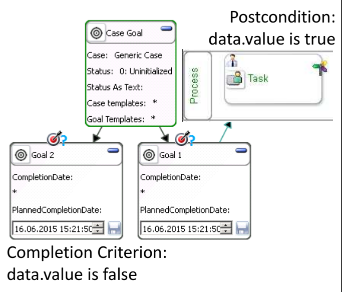
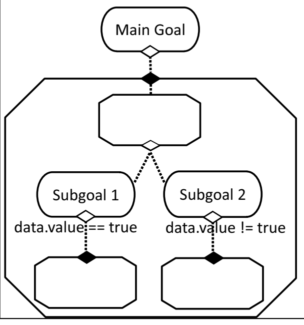
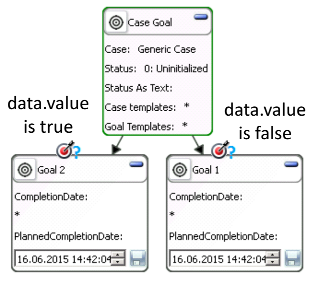
Table 1. Loop-Related Inconsistencies in ACM

Naming	**CMMN Example**	**ISIS Papyrus Example**
Inaccessible Dependency Loop		
Potential Endless Loop		Not possible (due to hierarchical organization of goals)

dependencies are generally inaccessible, then we call this kind of inconsistency *Inaccessible Dependency Loop*. In the given example in Table 1, *Task 1* is dependent on *Task 2* and *Task 2* is dependent on *Task 1*. Neither one is accessible due to this arrangement. Another inconsistency related to loops is called *Potential Endless Loop*, and it is found in parts of the model where loops do not include user interaction. In the example in Table 1 there is a dependency from *Goal 1* to *Goal 2* and another dependency from *Goal 2* to *Goal 1*. Because of the lack of user interactions in between these goals, there is a possibility that the case execution engine continuously reenters *Goal 1* and *Goal 2* until an action independent from this loop changes data states in a way to eventually break the loop.

Pre- and postconditions of tasks, completion criteria of goals, and dependencies among case elements are important and recurring mechanisms in ACM that guard the access to certain elements of a case, dependent on the case's data state. Table 2 summarizes inconsistencies in ACM definitions that are related to guarding constraints. If we observe just a single constraint, this isolated constraint must be inherently consistent. A simple example for a *Self-Contradictory Constraint* is `data.value == true && data.value != true` because the logical and-connected parts of this constraint can never be satisfied at the same time. We reuse this straightforward example of an unsatisfiable combination of constraints for the other possible inconsistencies that are shown in Table 2. *Directly Dependent Contradictory Constraints* occur if a dependency is defined between two constraints that contradict each other. *And-Dependent Contradictory Constraints* are contradictory constraints that are not explicitly connected by a single dependency but by an arrangement that links them together. Table 2 illustrates such an arrangement where the postconditions of both *Task 1.1* and *Task 1.2* must be met in order to allow a start of *Task 2*. Constraints in subprocesses or substructures can be contradictory to constraints of their surrounding structures. Table 2 illustrates an example of *Substructure- or Subprocess-induced Contradictions* in which the guard condition that is defined on the sequence flow in front of the end event contradicts the completion criterion of the goal. Regularly, there exist larger structures of interdependent constraints that can be composed of the aforementioned structures. An example for an *Unsatisfiable Composition of Interdependent Constraints* is given in Table 2 in which the *Main Goal* has a constraint structure that contains contradictory constraints. In particular, the postcondition of *Task 1* contradicts the completion criterion of *Goal 2*. Such compositions can comprise several aforementioned structures. In the example, directly dependent constraints are combined with and-dependent constraints in a big-

Table 2. Constraint-Related Inconsistencies in ACM

Naming	**CMMN Example**	**ISIS Papyrus Example**
Self-Contradictory Constraint	 Task Exit Criterion: data.value == true && data.value != true	 Task Postconstraint: data.value is true and data.value is false
Directly Dependent Contradictory Constraints	 Task 1 Task 2 Exit Criterion: data.value != true Entry Criterion: data.value == true	 Task 1 Task 2 Postconstraint: data.value is false Preconstraint: data.value is true
And-Dependent Contradictory Constraints	 Exit Criterion: data.value == true Task 1.1 Task 2 Task 1.2 Exit Criterion: data.value != true	 Postconstraint: data.value is true Task 1.1 Task 2 Task 1.2 Postconstraint: data.value is false
Substructure- or Subprocess-induced Contradictions	 Subprocess Task Guard Condition: data.value == true Entry Criterion: data.value != true Goal	 Process Task Guard Condition: data.value is true Completion Criterion: data.value is false Process Goal CompletionDate: PlannedCompletionDate: 16.06.2015 10:10:35
Unsatisfiable Composition of Interdependent Constraints	 Exit Criterion: data.value == true Main Goal Task 1 Goal 2 Goal 1 Entry Criterion: data.value != true	 Case Goal Case: Generic Case Status: 0: Uninitialized Status As Text: Case templates: * Goal Templates: * Postcondition: data.value is true Process Task Goal 2 Goal 1 CompletionDate: PlannedCompletionDate: 16.06.2015 15:21:50 16.06.2015 15:21:50 Completion Criterion: data.value is false
Conflicting Goals in Goal Hierarchies	 Main Goal Subgoal 1 Subgoal 2 data.value == true data.value != true	 Case Goal Case: Generic Case Status: 0: Uninitialized Status As Text: Case templates: * Goal Templates: * data.value is true data.value is false Goal 2 Goal 1 CompletionDate: PlannedCompletionDate: 16.06.2015 14:42:04 16.06.2015 14:42:04

ger, composite structure. Goals can be structured hierarchically [6, 14]. On top of the hierarchy stands the main goal of a case which is subdivided into subgoals. In Table 2, an example of a goal hierarchy is given. Naturally, pursuing contradictory goals simultaneously would not make any sense. Knowledge workers can work towards *Subgoal 1* and *Subgoal 2* concurrently, so their completion criteria must not be contradictory.

3 Structural Consistency Checking Approach for ACM

This section introduces our proposed approach for enabling structural consistency checking in ACM. Our approach leverages model checking and graph algorithms.

Model checking performs an exhaustive check whether a model meets a given specification. A model is often defined as a state transition system, whereas specifications are usually defined in a formal language such as Computation Tree Logic (CTL). The CTL expression $EF\ p$ is satisfied if the expression p holds on at least one subsequent path. Since model checking works faster when the state space is smaller, it is a good strategy to look at isolated parts of the model first (Definitions 2, 3 and 4).

Definition 2. (Self-Contradictory Constraint) *A constraint $c \in \mathcal{J}$ is self-contradictory iff* $\neg\left(EF\left(c\right)\right)$.

Definition 3. (Directly Dependent Contradictory Constraints) *Two constraints $c \in \mathcal{C}$ and $c' \in \mathcal{C}$ which are directly linked by a dependency $(c, c') \in \mathcal{D}_{\mathcal{X}\mathcal{E}} \cup \mathcal{D}_{\mathcal{S}_{\mathcal{X}}} \cup \mathcal{D}_{\mathcal{S}_{\mathcal{E}}}$ are contradictory iff* $\neg\left(EF\left(e \wedge e'\right)\right)$.

Definition 4. (And-Dependent Contradictory Constraints) *Two constraints $c \in \mathcal{C}$ and $c' \in \mathcal{C}$ which are and-dependent because $\exists((c,t) \wedge (c',t')) \mid (c,t) \in \mathcal{D} \wedge (c',t') \in \mathcal{D} \wedge t = t' \wedge c \in \mathcal{C} \wedge c' \in \mathcal{C} \wedge t \in \mathcal{C} \wedge t' \in \mathcal{C}$ are contradictory iff* $\neg\left(EF\left(e \wedge e'\right)\right)$.

The remainder of this section is concerned with finding contradictions in larger structures. We focus exemplary on *Dependency Loops* and *Goal Hierarchies*.

3.1 Dependency Loops

Definition 5. (Case Graph) *$G_{\mathcal{M}} = (V, E)$ is a directed graph representation of $\mathcal{M}$, where $V = \mathcal{T} \cup \mathcal{G} \cup \mathcal{S} \cup \mathcal{C} \cup \mathcal{D}$ is a set of vertices. E is a set of edges which is created as follows:*

- *An edge $\left(e, \zeta_{\mathcal{E}}\left(e\right)\right)$ is added for every $e \mid e \in \mathcal{E}$.*
- *An edge $\left(\zeta_{\mathcal{X}}\left(x\right), x\right)$ is added for every $x \mid x \in \mathcal{X}$.*
- *An edge (f, d) is added for every $d = (f, t) \in \mathcal{D}$.*
- *An edge (d, t) is added for every $d = (f, t) \in \mathcal{D}$.*

Definition 6. *A dependency loop $\circlearrowleft$ is given as a strongly connected component (SCC) of $G_{\mathcal{M}}$.*

The set of strongly connected components can be computed by Tarjan's algorithm [15].

Definition 7. (Inaccessible Dependency Loop) *A dependency loop* $\circlearrowleft$ *is inaccessible iff* $(\nexists e \mid e \in \mathcal{E} \wedge \zeta_{\mathcal{E}}(e) \in \circlearrowleft \wedge e \notin \circlearrowleft) \wedge (\nexists d = (f, t) \mid d \in \mathcal{D} \wedge d \notin \circlearrowleft \wedge t \in \circlearrowleft \wedge t \in T \cup S)$

According to this definition, a dependency loop is accessible if there exists at least either

- one entry criterion which is attached to an element of the dependency loop but is not part of the loop, or
- a dependency which points to a task or stage of the loop without being in the loop.

3.2 Goal Hierarchies

For the verification of hierarchically structured goals, we first find pairs of goals which are possibly in contradiction to each other.

Definition 8. *A goal* $g \in \{x \mid x \in \mathcal{G} \wedge \exists \eta(x)\}$ *is possibly in contradiction with a goal* g' *iff* $g' \in \mathcal{G} \setminus \mathcal{G}_\eta$ *where* $\mathcal{G}_\eta$ *is the set of parent goals of* g *which can be found by recursive use of* η *(i.e.,* $\eta(g)$*,* $\eta(\eta(g)), \eta(\eta(\eta(g))), \ldots$*) until* η *returns no further parent goal.*

Then, we can check each found possibly contradictory goal pair (g, g'). Let $g.\mathcal{E}$ denote the set of entry criteria of g and $g'.\mathcal{E}$ be the set of entry criteria of g'.

Definition 9. *(Conflicting Goals in Goal Hierarchies) Two goals* g *and* g' *are contradictory iff they are possibly in contradiction to each other (cf. Definition 8) and the formula* $EF(e \wedge e')$ *does not hold for at least one* $\{e, e'\} \mid e \in g.\mathcal{E} \wedge e' \in g'.\mathcal{E}$.

4 Experimental Results

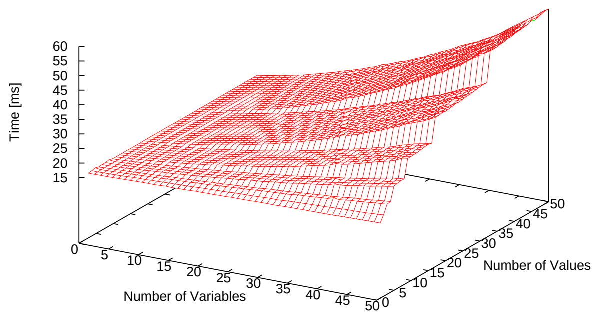


Fig. 1. Results of performance and scalability evaluation

Model checking is known for its high demands on computational resources. Therefore, we conducted an experiment on the performance and scalability of our approach.

In this experiment, we measure the response times that are to be expected while checking a combination of string constraints with the NuSMV model checker version 2.5.4 [2]. In particular, an *and-connected* constraint term that contains between 1 and 50 string variables is created. Furthermore, the number of possible string values of each variable is between 1 and 50. This scenario is representative for checking large structures of interdependent constraints in ACM systems because a high count of interdependent constraints is to be expected. The experiment was carried out on a computer with 8 GB RAM, Intel i5-4200U CPU and SATA II SSD, as we wanted to test our approach in the usual setting of a software developer or knowledge worker. The experiment was repeated 1000 times which resulted in sufficient data. The reported values shown in Figure 1 represent the median of our measurements which is stable against statistical outliers (e.g., caused by OS activities that we cannot control). Studying the averages yields similar results and the standard deviation is not significant. Even with a large number of variables and values, the response time of the model checker stays far below 100 ms, which is still reasonable for checking large problems.

5 Related Work

Formal verification encompasses techniques such as model checking or Petri net based approaches to assert whether a model definition satisfies certain properties. Kherbouche et al. use the SPIN model checker[5] to find structural errors in BPMN 2 models [7]. Eshuis proposes a model checking approach using the NuSMV model checker for the verification of UML activity diagrams [5]. Van der Aalst defines a mapping of EPCs to Petri nets for checking structural errors [1]. Sbai et al. use SPIN for the verification of workflow nets [12]. Raedts et al. propose the transformation of models such as UML activity diagrams and BPMN 2 models to Petri nets for verification with Petri net analyzers [10]. Köhler et al. describe a process by means of an automaton and check this automaton by NuSMV [8]. El-Saber et al. [4] provide a formalization of BPMN and translate BPMN constructs to the Maude model checker [3]. None of the existing approaches proposes structural consistency checking for ACM.

6 Conclusion and Future Work

This paper discusses structural consistency checking of ACM. We discuss structural concepts of ACM and what inconsistencies can possibly occur in such structures. As our main contribution, we propose structural consistency checking of ACM definitions that leverages model checking and graph algorithms. Our approach is able to discover various structural inconsistencies in ACM cases. The experimental results show that our consistency checking scales well. Our approach strives for computational efficiency despite the use of model checking. This is achieved by dividing the problem of checking structural consistency of entire case models into smaller problems. As a result, if already a single isolated constraint is self-contradictory, it is not necessary to check it as part of a bigger structure that demands greater computational effort. The approach has been

[5] `http://spinroot.com`

developed in a prototype for an industrial system and tested with real cases.

Acknowledgement. The research leading to these results has received funding from the FFG project CACAO, no. 843461 and the WWTF project CONTAINER, Grant No. ICT12-001.

Bibliography

[1] Van der Aalst, W.M.P.: Formalization and verification of event-driven process chains. Information and Software Technology 41(10), 639 – 650 (1999)

[2] Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NuSMV: a new Symbolic Model Verifier. In: 11th Conf.on Computer-Aided Verification (CAV). pp. 495–499. Springer (July 1999)

[3] Eker, S., Meseguer, J., Sridharanarayanan, A.: The Maude LTL Model Checker and Its Implementation. In: 10th Intl. Conf. on Model Checking Software (SPIN). pp. 230–234. Springer (2003)

[4] El-Saber, N., Boronat, A.: BPMN formalization and verification using Maude. In: 2014 Workshop on Behaviour Modelling-Foundations and Applications (BM-FA). pp. 1:1–1:12. ACM (2014)

[5] Eshuis, R.: Symbolic model checking of UML activity diagrams. ACM Trans. Softw. Eng. Methodol. 15(1), 1–38 (2006)

[6] Greenwood, D.: Goal-oriented autonomic business process modeling and execution: Engineering change management demonstration. In: 6th Intl. Conf. BPM. pp. 390–393. Springer (2008)

[7] Kherbouche, O., Ahmad, A., Basson, H.: Using model checking to control the structural errors in BPMN models. In: 7th Intl. Conf. on RCIS. pp. 1–12 (2013)

[8] Koehler, J., Tirenni, G., Kumaran, S.: From business process model to consistent implementation: a case for formal verification methods. In: 6th Intl. Conf. on EDOC. pp. 96–106 (2002)

[9] Pucher, M.J.: Considerations for implementing adaptive case management. In: Fischer, L. (ed.) Taming the Unpredictable Real World Adaptive Case Management: Case Studies and Practical Guidance. Future Strategies Inc. (2011)

[10] Raedts, I., Petković, M., Usenko, Y.S., van der Werf, J.M., Groote, J.F., Somers, L.: Transformation of BPMN models for Behaviour Analysis. In: MSVVEIS. pp. 126–137. INSTICC (2007)

[11] Rinderle-Ma, S., Reichert, M., Dadam, P.: Correctness criteria for dynamic changes in workflow systems: A survey. Data & Knowledge Engineering 50(1), 9 (2004)

[12] Sbai, Z., Missaoui, A., Barkaoui, K., Ben Ayed, R.: On the verification of business processes by model checking techniques. In: 2nd Intl. Conf. on ICSTE. vol. 1, pp. 97–103 (Oct 2010)

[13] Schonenberg, H., Mans, R., Russell, N., Mulyar, N., van der Aalst, W.M.P.: Process flexibility: A survey of contemporary approaches. In: CIAO and EOMAS. pp. 16–30. Springer (2008)

[14] Stavenko, Y., Kazantsev, N., Gromoff, A.: Business process model reasoning: From workflow to case management. Procedia Technology 9(0), 806 – 811 (2013)

[15] Tarjan, R.: Depth first search and linear graph algorithms. SIAM Journal on Computing (1972)