Towards Conflict-Free Composition of Non-functional Concerns

Benjamin Schmeling[1,2], Anis Charfi[1], Marko Martin[1], and Mira Mezini[2]

[1] Software Engineering & Tools, SAP Research Darmstadt, Germany
firstname.lastname@sap.com
[2] Technische Universität Darmstadt, Darmstadt, Germany
mezini@informatik.tu-darmstadt.de

Abstract. In component-based software development, applications are decomposed, e.g., into functional and non-functional components which have to be composed to a working system. The composition of non-functional behavior from different non-functional domains such as security, reliability, and performance is particularly complex. Finding a valid composition is challenging because there are different types of interdependencies between concerns, e.g. mutual exclusion, conflicts, and ordering restrictions, which should not be violated.

In this paper we formalize a set of interdependency types between non-functional actions realizing non-functional behavior. These interdependencies can either be specified explicitly or implicitly by taking action properties into account. This rich set of interdependencies can then be used to ease the task of action composition by validating compositions against interdependency constraints, proposing conflict resolution strategies, and by applying our guided composition procedure. This procedure proposes next valid modeling steps leading to conflict-free compositions.

Keywords: Feature Interaction, NFC Composition, Model-driven development, Web Services.

1 Introduction

Non-functional behavior — in contrast to functional behavior — does not provide consumable business functionality. The execution of non-functional behavior rather improves the quality of a software system by satisfying certain non-functional attributes. For example, an encryption algorithm is a behavior which can be executed in order to support confidentiality. Functional behavior is often encapsulated by the main language constructs of the underlying programming platform such as operations or classes in case of object-oriented languages whereas non-functional behavior is hard to modularize by these constructs. Hence, language extensions have been introduced such as aspect-oriented programming which offers aspects to complement the deficiency of modularizing crosscutting concerns.

J. Ralyté et al. (Eds.): CAiSE 2012, LNCS 7328, pp. 80–94, 2012.

In [15] and [16], we introduced a model-driven approach for composing non-functional concerns in web services. This approach supports web services (black box view) and composite web services (gray box view) as well as the specification and enforcement of non-functional concerns. It is based on a well-defined process with different phases: requirements specification, action definition, action composition, action-to-service mapping, action-to-middleware-service mapping, and generation of NFC enforcement code. In our approach, the non-functional behavior is represented by non-functional actions (NFAs), defined in the action definition phase, which are supporting certain non-functional attributes defined in the requirements specification phase. Non-functional behavior is often non-orthogonal, i.e., when there are multiple actions activated simultaneously, a certain execution order has to be respected (cf. Beauvois [1]). More generally, there are different types of interdependencies (aka. interactions as motivated by [14], [11], [4]) between NFAs, e.g., mutual exclusion, data dependencies, and ordering restrictions. These interdependencies are constraints upon the execution of behavior. Consequently, a set of NFAs should constitute a well-defined execution order which can be defined in the action composition phase. The action compositions can then be mapped to web services in the mapping phase and related to concrete middleware services implementing the runtime behavior of NFAs. This allows to generate code for enforcing the composition at runtime.

In our previous works, we have already defined the modeling framework for the black and gray box view and provided a runtime environment based on proxies to enforce the models at runtime. In this paper, we complement this work by adding tool support to the complex task of action composition. This composition is modeled by domain experts; however, knowledge from different non-functional domains is required in order to understand the impact of different NFAs. This knowledge should be captured by a reusable model and provided to the domain experts. The contributions of this paper are thus more specifically:

- Defining a formal interdependency model that helps to identify invalid composition definitions at design time.
- Enriching this model by discovery of cross-domain interdependencies through analysis of the data impact of NFAs.
- Using the interdependency model to provide support for composing NFAs by:
  - visualizing constraint violations in the composition,
  - suggesting conflict resolution strategies for violated constraints,
  - introducing a guided modeling procedure.

The structure of the paper is as follows. In Section 2, our interdependency model is introduced and additional properties are described which can be used to discover implicit interdependencies. Section 3 describes our solution for modeling conflict-free compositions of non-functional behavior. In Section 4, the implementation of the modeling tool is described. Furthermore, our approach is evaluated by instantiating it in the web services context. Section 5 analyzes related work and Section 6 concludes this paper.

2 Formalizing the Interdependency Model

2.1 Tasks and Actions

Let $\mathcal{A}$ be a set of NFAs, and let $\mathcal{T}$ be tasks each executing an NFA. We define $executes \subseteq \mathcal{T} \times \mathcal{A}$ as the relation defining which task executes which action. In contrast to actions, tasks are part of a specific execution context (i.e., a process) and therefore have a well-defined execution order. This relation can be compared to the relation between BPMN [12] service tasks and the services called by these tasks. More details of the composition model can be found in Section 3.1.

2.2 Interdependencies

We define $\mathcal{I} := mutex \cup requires \cup precedes$ as the set of interdependencies between actions and tasks. More specifically it is:

- $requires = \{(x_1, x_2) \in (\mathcal{A} \times \mathcal{A}) \cup (\mathcal{T} \times \mathcal{T}) \mid$ Execution of x_1 requires the execution of $x_2\}$
- $precedes = \{(x_1, x_2) \in (\mathcal{A} \times \mathcal{A}) \cup (\mathcal{T} \times \mathcal{T}) \mid$ If x_1 and x_2 are both executed, x_1 has to be executed before $x_2\}$
- $mutex = \{(x_1, x_2) \in (\mathcal{A} \times \mathcal{A}) \cup (\mathcal{T} \times \mathcal{T}) \mid$ Execution of x_1 excludes the execution of $x_2\}$

From a given set of interdependencies, further interdependencies can be inferred by symmetry and transitivity. *Mutex* is symmetric, i.e., $mutex(x_1, x_2) \rightarrow mutex(x_2, x_1)$. *Requires* and *precedes* are both transitive, i.e., $precedes(x_1, x_2) \wedge precedes(x_2, x_3) \rightarrow precedes(x_1, x_3)$ (where $x_1, x_2, x_3 \in \mathcal{A} \cup \mathcal{T}$). In addition to these interdependencies, there are also action properties that play an important role for the composition. These properties can be categorized into data-related properties and control-flow-related properties (not in the scope of this paper).

2.3 Data Dependencies

Let $\mathcal{A}$ be a set of actions and $\mathcal{D}$ be a set of data items (which can be of complex type) and $a \in \mathcal{A}$ and $d \in \mathcal{D}$. Then, $\mathcal{P} := \{read, add, remove, modify\}$ is the set of binary relations between actions and data which we call impact types (because they define the impact on data) with the following semantics:

- $read = \{(a, d) \in \mathcal{A} \times \mathcal{D} \mid$ a reads data item d$\}$
- $add = \{(a, d) \in \mathcal{A} \times \mathcal{D} \mid$ a adds data to data item d$\}$
- $remove = \{(a, d) \in \mathcal{A} \times \mathcal{D} \mid$ a removes data item d$\}$
- $modify = \{(a, d) \in \mathcal{A} \times \mathcal{D} \mid$ a modifies (and reads) data item d$\}$

Let a and b be actions accessing data item d and let a be executed directly before b, i.e., there is no other action $c \neq a, b$ accessing d executed between a and b. Then, there are 16 possible combinations of impact types to be analyzed. We found 10 combinations that cause or might cause conflicts w.r.t. their impact on data as shown in Table 1.

Table 1. Conflicting combinations of impact types: − conflict, (−) potential conflict, + no conflict, (+) warning, R = reverse order also in conflict. Subscripted numbers indicate the number of the enumeration item which explains the respective conflict.

	read(b,d)	**add(b,d)**	**remove(b,d)**	**modify(b,d)**
read(a,d)	+	$-_2$	+	+
add(a,d)	+	$-_{6,R}$	$(+_5)$	+
remove(a,d)	$-_1$	+	$-_{7,R}$	$-_{9,R}$
modify(a,d)	$(-_4)$	$-_3$	$(+_{8,R})$	$(-_{10,R})$

1. $remove(a, d) \wedge read(b, d)$ Data d is removed by a before b is able to read it.
2. $read(a, d) \wedge add(b, d)$ Data d is read by a before b adds it. Either d exists before execution of a which would lead to duplicated data by the execution of b, or d has to be added by b because it does not exist, so a would read non-existing data.
3. $modify(a, d) \wedge add(b, d)$ Data d is modified by a before b adds it. This combination is similar to 2, because $\forall a \in \mathcal{A}.modify(a, d) \rightarrow read(a, d)$.
4. $modify(a, d) \wedge read(b, d)$ Data d is modified by a before b can read it. This might be a potential conflict because a modifies the data which causes a state change from d_1 to d_2. It depends which state b expects. If b expects d to be in state d_1, this combination is a conflict.
5. $add(a, d) \wedge remove(b, d)$ Data d is added by a and then removed by b. Removing data directly after adding it makes no sense, but does not cause problems at runtime.
6. $add(a, d) \wedge add(b, d)$ Data d is added by action a and b and hence duplicated.
7. $remove(a, d) \wedge remove(b, d)$ Data d is removed by action a and b. After execution of a, data d does not exist anymore; hence b cannot remove it.
8. $modify(a, d) \wedge remove(b, d)$ Data d is modified by a and then removed by b. Removing data directly after modifying it makes no sense, but does not cause problems at runtime.
9. $remove(a, d) \wedge modify(b, d)$ Data d is removed by action a and then modified by b. This is similar to 1 because $\forall a \in \mathcal{A}.modify(a, d) \rightarrow read(a, d)$.
10. $modify(a, d) \wedge modify(b, d)$ Data d is modified by a and then modified again by b. As modify also reads data, this is similar to 4.

We will discuss strategies for resolving these conflicts in the following. Since the strategies can only be applied when the execution order of actions is already known, we assume tasks x and y executing conflicting actions a and b which access the same data item d such that x is executed before y and there is no task z between them which also accesses d. The first five conflict situations are resolvable by inverting the execution order of tasks x and y because the reverse order causes no data conflicts as can be seen in Table 1. For those combinations, we add the *precedes* interdependency to the set of interdependencies, i.e., $precedes(y, x)$ in this case. All other combinations cannot be resolved by reordering because the inverse order might also cause conflicts. Those conflicts can be resolved either by removing one of the tasks or by executing them exclusively, i.e., by executing

either x or y depending on some condition. For all those combinations, we add *mutex* to the set of the existing interdependencies, i.e., $mutex(x, y)$ in this case.

3 Towards Conflict-Free Action Compositions

3.1 The Composition Model

For the composition of actions, we use a subset of BPMN2 [12]. A model in BPMN2 is a set of nodes and transitions between them. We define our simplified process model as follows. A non-functional activity is a directed graph containing all process elements. We define $activity := (\mathcal{N}, \mathcal{E})$ with the following semantics:

$$\begin{aligned}
\mathcal{N} :=& \{x | node(x)\} \equiv \{x | x \text{ is process node}\} \\
\mathcal{E} :=& \{(x, y) | transition(x, y)\} \\
\equiv& \{(x, y) \in \mathcal{N} \times \mathcal{N} | \text{there is a transition from x to y}\} \\
start :=& \{x | x \text{ is the start node of the process}\} \\
end :=& \{x | x \text{ is an end node of the process}\} \\
\mathcal{T} :=& \{x | task(x)\} \equiv \{x | x \text{ is task node}\} \subset \mathcal{N} \\
\mathcal{G} :=& \{x | gateway(x)\} \equiv \{x | x \text{ is gateway node}\} \subset \mathcal{N} \\
\mathcal{XOR} :=& \{x | gw_xor(x)\} \equiv \{x | x \text{ is xor gateway}\} \subseteq \mathcal{G} \\
\mathcal{OR} :=& \{x | gw_or(x)\} \equiv \{x | x \text{ is or gateway}\} \subseteq \mathcal{G} \\
\mathcal{AND} :=& \{x | gw_and(x)\} \equiv \{x | x \text{ is and gateway}\} \subseteq \mathcal{G} \\
\mathcal{M} :=& (\mathcal{T}, \mathcal{XOR}, \mathcal{OR}, \mathcal{AND}, start, end)
\end{aligned}$$

$\mathcal{M}$ is a tuple of sets $M_0, M_1 ...$ and each node n is exactly in one of its set elements: $(\forall i, j < |\mathcal{M}|) M_i \cap M_j = \emptyset$ for $i \neq j$, and $(\forall n \in \mathcal{N})(\exists i) n \in M_i$. Moreover, there is exactly one start node: $|start| = 1$.

3.2 Identifying Constraint Violations

To identify violations of the given interdependency constraints, we have to check a non-functional activity against each individual interdependency. For each interdependency $i = (a, b) \in \mathcal{I}$, the occurrence and order of actions (or tasks) a and b in the same execution path can lead to constraint violations depending on the given type. Hence, all possible execution paths through the process graph have to be analyzed. The number of those paths depends on the control flow of the process, more specifically on the number of OR and XOR gateways and the number of outgoing sequence flows per gateway. Presuming all gateways are used in sequence and $out : \mathcal{G} \rightarrow \mathbb{N}$ is a function that calculates the number of outgoing sequence flows for each gateway, the number of possible paths for a given number of XOR and OR gateways, respectively, in the worst case is:

$$paths_{xor} = \prod_{x \in \mathcal{XOR}} (out(x)) \qquad paths_{or} = \prod_{x \in \mathcal{OR}} (2^{out(x)})$$

Obviously, the number of possible paths for *OR* is much higher than that of *XOR*. Let k be the number of all outgoing sequence flows from *OR* gateways and *traverse* be a function which traverses all possible paths, then the complexity of this function can be estimated using the $\mathcal{O}$-Notation: $traverse \in \mathcal{O}(2^k)$ resulting in exponential runtime complexity. Since there are already existing methods for searching defined spaces for possible solutions, we decided to leverage those solutions. We make use of the declarative Prolog language because it provides very efficient ways to do depth-first search with backtracking.

3.3 Using Prolog to Find Violations and Counter Examples

In order to use Prolog for constraint checking, we have to import our BPMN models into Prolog. This can be achieved by transforming activities into a fact data base which contains a set of predicates. Hence, the following predicates have been defined: *start(x)*, *end(x)*, *task(x)*, *gw_ xor(x)*, *gw_ or(x)*, *gw_ and(x)*, *node(x)*, *transition(x,y)*, and *executes(x, a)*. The last predicate defines that the task node x of the process executes action a. We distinguish between tasks and actions in order to cope with processes in which different task nodes execute the same action. The constraints given by the interdependency model form the rules that should apply to the fact base. However, the challenge for defining these rules was that the violation of some rule should not result in a simple boolean true or false decision, but should also provide some counter examples to give the modeler of non-functional activities constructive feedback.

With its backtracking concepts, Prolog allows for obtaining all values for which a certain predicate evaluates to true. Therefore, we defined a predicate with parameters A, B, X, Y, and P for each interdependency type so that the predicates are true if and only if P is a counter example for the respective interdependency regarding the actions A and B which are executed in nodes X and Y, respectively. Within a query, Prolog distinguishes between constants and variables: If a variable is used for a certain parameter of a predicate, Prolog will search for values of this variable fulfilling the predicate whereas constant parameters restrict the search space. We may assume to have a constant list of interdependencies between actions and tasks. In case of an action interdependency, we can just apply the appropriate predicate for the respective interdependency type to constants for A and B and variables X, Y, and P for Prolog to yield possible counter examples as solutions for X, Y, and P. In case of a task interdependency, the procedure is analogous, but then we use the task constants for X and Y.

Regarding the structural representation of counter examples, we introduced the following concepts of paths in BPMN processes: A **Plain Graph Path** (PGP) from X to Y is a simple path from X to Y in the BPMN process graph as known from graph theory of directed graphs. It is represented as the list of nodes contained in the path. A **Block Path** (BP) is a PGP where nodes between opposite gateways are left out. BPs can only exist between two nodes if they have the same parent node. The term *parent node* refers to a tree representation of the BPMN process nodes in which the parent node of each node is the gateway

in which it is contained, or, if it is not contained in any gateway, an imaginary root node. For each pair of nodes X, Y with the same parent node, there is exactly one BP between these nodes if a PGP from X to Y exists. A **Block Execution Path** (BEP) is a BP where each gateway node is replaced with a pair (X, P). X is the replaced gateway node itself, and P is a list of BEPs that are executed in parallel starting from the gateway X. BEPs respect gateway semantics, e.g., the number of paths starting from an XOR gateway is always 1. A BEP is therefore an appropriate representation of a concrete execution of the BPMN process. Particularly, BEPs are used to represent counter examples in the aforementioned rules. A BEP is called complete if it begins with a start node and ends with an end node. Specifically, we defined the following rules for the interdependency types, each starting with *ce* as an abbreviation for *counter example*. Lets assume that P is a complete BEP, then it is:

- $ce\_conflicts(A, B, X, Y, P)$ is true if P contains both X and Y which in turn execute the actions A and B, respectively.
- $ce\_precedes(A, B, X, Y, P)$ is true if both action A and action B are executed in P, but task Y which executes B is in that path not guaranteed to be preceded by another task which executes A. X is just any task executing A in this path. Intuitively, this predicate is true if action A is not guaranteed to be executed before B. This is the case if B appears sequentially before the first A, or if A and B are executed in parallel paths of the same gateway.
- $ce\_requires(A, B, X, Y, P)$ is true if A is executed by X in P, but there is no task executing B. By convention, we set Y to 0.

Each of the predicates can be used to obtain counter examples by defining constants for A and B and using variables for X, Y, and P for which Prolog will try to find instances which make the predicate true. For that purpose, Prolog iterates over all complete BEPs and tasks X, Y executing A, B and returns the first combination fulfilling the respective predicate. If no counter example exists, no solution will be found.

3.4 Conflict Resolution

After identifying interdependency violations in a non-functional activity, strategies for solving these conflicts should be defined. Let us assume that $(a, b) \in \mathcal{I}$ is a violated task or action interdependency. Table 2 shows the different strategy classes. The difference between *rearrange* and *move* is that with *rearrange* an action will not move from one execution branch to another. As can be seen in the table, all resolution strategies might impose new conflicts. To avoid these undesired side effects we analyzed under which conditions a certain strategy can be applied safely. Due to space limitations we only give a short example for the *remove action* strategy. Before *remove action* can be applied safely to action a, for instance, we just have to check if there is a *requires* interdependency from any other action to a. In general, we distinguish both safe and potentially unsafe strategies.

Table 2. Resolution strategies: Interdependency conflicts solved and introduced

Resolution Strategy	Solves	Might Introduce
Remove Action	Mutex, Prec	Req
Insert Action	Req, Prec	Prec, Mutex
Rearrange Action	Prec	Prec
Move Action	All	All
Transform Gateway	All	All

3.5 Conflict-Free Composition Procedure

As we have seen in the previous subsection, it is complex to provide a validation mechanism with automatic conflict resolution. Mostly, some kind of human intervention is required at some point. It is even harder to propose a complete conflict-free activity because of the possibly small sets of given interdependencies. In order to combine the power of our validation approach with the ability of human non-functional domain experts to compose activities, we propose to use a guided modeling procedure. The idea is that a composition tool can be used to model a start event and the tool then proposes the next valid steps leading always to correct processes w.r.t. interdependency constraints. For that purpose, we had to extend our Prolog implementation in the following way: It should take a predefined set of candidate actions and the BPMN node from where to insert the next action as input. The output should be a list of valid actions which, when inserted at this point, would not cause any interdependency violation.

The concrete process for obtaining a list of valid actions A to be proposed for insertion at a certain position consists of the following steps: (1) Virtually extend the current (incomplete) BPMN process by adding a placeholder task x at the position where the user wants to insert a new element. Also, for each node of the process without an outgoing edge, add an edge to the end event which is newly created if necessary. This allows to have a complete BPMN process enclosed by a start and an end event which our Prolog program is able to process. (2) Send a query to Prolog to obtain all actions I which would violate a constraint if they were executed by x. This query is based on the Prolog model of the BPMN process such as used during validation and, additionally, on the Prolog model of all interdependencies relevant for the BPMN process. These are expressed in terms of a list of Prolog facts based on Prolog predicates *precedes*, *requires*, and $conflicts$, each having two parameters defining the actions or tasks between which the respective interdependency exists. The query also contains the list of candidate actions C to be tested at the position of x. (3) Our Prolog program then consecutively assumes x to execute each of the candidate actions and returns the list I of them for which at least one of the defined interdependencies is violated. (4) The actions $A := C \setminus I$ are proposed to the user for insertion at the specified position. As, by definition of the proposed actions A, the obtained process after insertion of one of them would never result in a new constraint violation, we state that this composition technique considerably facilitates the definition of conflict-free compositions.

4 Running Example and Implementation

In [15] and [16], we introduced a new methodology for composing non-functional concerns in web services. This methodology proposes an engineering process with different phases. These phases are in general (with slight modifications) applicable to all kinds of software systems since they do not assume any concrete technology. However, the mapping phase is web-service-specific because actions and activities are mapped to concrete web services. In the following, we will use web services as a concrete technology for applying our abstract approach in order to show the instantiation of the interdependency model and to prove the applicability of the conflict detection and our solution strategy. Thereby, we will adhere to a part of the modeling procedure from our previous works as presented in the introduction, namely requirements definition, action definition, and action composition. For each of these phases — and also for the action-to-service mapping which is not in the focus of this paper — a separate editor has been implemented within the Eclipse IDE using the Graphiti framework[1].

Let us assume an enterprise which has transformed its IT assets into a set of commercial web services. We further assume these web services have been implemented in different programming languages, e.g., due to constraints introduced by some legacy systems. During the development of the web service, non-functional concerns have been ignored on purpose: they should be strictly separated from the business functionality of the services. Hence, depending on the respective features the service provides, different non-functional requirements have been identified. For example, since the services are commercial, authentication and authorization are required to restrict the access to the services only to registered customers. To bill the customers based on the service usage, an accounting mechanism is required as well as support for non-repudiation and integrity of the messages. Furthermore, the company decided to log messages in the early introduction phase and to monitor the response time of their services. Another requirement is that the response time should be as low as possible.

The security, billing/accounting, performance, monitoring/logging, and general web service experts of the company transform the requirements into non-functional actions capable of fulfilling the requirements. The security expert knows how to support the security requirements and defines the following actions using our action editor: *Authenticate* (for authenticity), *Authorize*, and *VerifySignature* (for non-repudiation and integrity). Having defined those actions, the security expert identifies possible interdependencies between them. He defines that *Authenticate* has to precede *Authorize* and that *Authorize* requires *Authenticate*. Furthermore, he uses XPath [5] expressions to describe the data items of the SOAP message which the actions have an impact on, e.g., *Authenticate* will read the *UsernameToken* which is part of the *Security* XML tag of the SOAP message header. A summary of all actions which have been defined by all experts can be found in Table 3. Table 4 (Column 1) summarizes all interdependencies that have been discovered by the experts. In the example, one

[1] `http://eclipse.org/`, `http://www.eclipse.org/graphiti/`

Table 3. Actions and their impact

Action	Expert	Impact (XPath)
Authenticate	Security	Read(/Header/Security/UsernameToken)
Authorize	Security	Read(/Header/Security/UsernameToken)
VerifySignature	Security	Read(/Header/Security/BinarySecurityToken, /Security/Signature)
RemSecHeaders	Security	Remove(/Header/Security)
Log	Log/Mon.	Read(/Message//*)
StartTimer	Log/Mon.	None
StopTimer	Log/Mon.	None
ReadFromCache	Perform.	Read(/Body//*)
RemoteAccounting	Acc./Bill.	Read(/Body//*)
LocalAccounting	Acc./Bill.	Read(/Body//*)
RemAllHeaders	General	Remove(/Header//*)

can see that most of the interdependencies are only discovered between actions defined by the same expert. Cross-concern interdependencies are only defined by the accounting expert who knows that due to legal issues he has to prove that a service invocation has really been caused by a certain customer. Cross-concern interdependencies are generally hard to identify because the experts have to understand and analyze all actions of all non-functional domains.

Table 4. Explicitly defined & implicit interdependencies

Explicit Interdependencies	Implicit Interdependencies
precedes(Authenticate, Authorize)	precedes(Authenticate, RemAllHeaders)
requires(Authorize, Authenticate)	precedes(Authorize, RemAllHeaders)
precedes(StartTimer, StopTimer)	precedes(VerifySignature, RemAllHeaders)
requires(StopTimer, StartTimer)	precedes(Log, RemAllHeaders)
requires(LocalAccounting, VerifySignature)	precedes(Log, RemSecHeaders)
requires(RemoteAccounting, VerifySignature)	mutex(RemSecHeaders, RemAllHeaders)
mutex(RemoteAccounting, LocalAccounting)	
precedes(Authenticate, RemSecHeaders)	
precedes(Authorize, RemSecHeaders)	
precedes(VerifySignature, RemSecHeaders)	

In the next modeling phase, the non-functional activity is created with the composition editor (shown in Figure 1) by importing the action definition and dragging the available actions from the palette into the activity. An action is executed by a special BPMN task (the arrow symbol) and additional gateways and sequence flow elements can be used to define the control flow.

When a concrete execution order at the task level is given, the previously defined interdependencies can be enriched by additional task interdependencies derived from the data dependencies: Possible data conflicts are identified by an

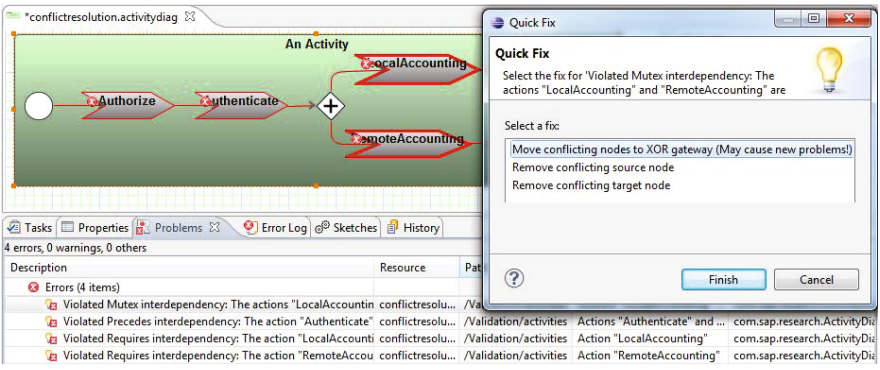

Fig. 1. Composition Editor: Conflict detection and resolution

intersection of the XPath expressions defining the data items that are affected by an action (shown in Table 3). If there is at least one node that both expressions have in common, the impact types are compared to each other. If, for instance, in the given process a task executing the *RemAllHeaders* action precedes another task executing an action accessing parts of the message header, there is a remove-read conflict between the two tasks. This data conflict can be resolved by introducing a precedes constraint upon these tasks. Another data conflict can be found when looking at the *RemAllHeaders* and the *RemSecurityHeaders* actions. The latter removes a subset of the data that *RemAllHeaders* removes. This is a remove-remove conflict which can be solved by introducing a mutex interdependency between the tasks executing those actions. The inferable interdependencies have been collected in Table 4 (Column 2).

Validation for a completely modeled action composition can be started by pushing the *Validate* button in the composition editor. Internally, the process and interdependency data, which is saved as an Ecore model, is transformed into Prolog facts and processed by our Prolog program. A list of all problems is shown in the problems view: a violation of *precedes* between *Authorize* and *Authenticate*, a violation of *mutex* between the accounting actions, and two *requires* violations due to the lack of *VerifySignature*. The selected problem is highlighted (see Figure 1). Moreover, so-called quick fixes are available via the context menu of each problem. In our example, the modeler can for example remove one of the tasks that are executing mutual exclusive actions or introduce an XOR gateway.

In the approach presented above the modeler gets feedback only when he triggers the validation. However, it is usually better to avoid those mistakes already during the modeling process. This is supported by our guided modeling procedure. Using this procedure, the user starts modeling and a context pad shows all available actions he can add next as shown in Figure 2. In the Graphiti-based context pad, the next valid actions are shown, e.g., after choosing the *LocalAccounting* action, the *RemoteAccounting* action is not available anymore except in another branch of an XOR gateway.

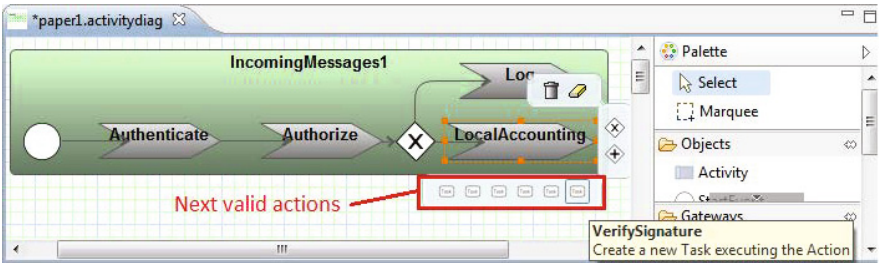

Fig. 2. Composition Editor: Guided composition proposes next valid actions

5 Related Work

5.1 General Approaches

The feature interaction problem, originally from the telecommunications domain, can also be found in object-oriented and component-oriented systems. For example, Pulvermüller et al. [13] define features as observable behaviors that can be of functional or non-functional nature. They distinguish unintended and intended feature interactions and interactions with positive and with negative effect. The feature interaction problem is similar to that of interdependencies between NFAs and also defines similar types of interactions.

Beauvois [1] defines composition constraints for *interweaving execution sequences* when composing non-orthogonal concerns. The author generates a scheduler from these constraints which is responsible for capturing all valid execution sequences in a dynamic way. The functional behavior as well as the composition constraints are described using an extension of hierarchical finite state machines. Beauvois describes constraints in form of state machines by describing the valid execution sequences explicitly whereas we use purely declarative constraints which fosters reuse and flexibility.

Sanen et al. [14] provide a conceptual model for concern interactions. They define a set of interaction types which we extended for our approach. Furthermore, they developed a prototype that automatically generates rules out of this model in order to improve concern interaction understanding. Those rules are used to build an expert system to support software engineers during development. The input to that expert system is a component composition specification and the output is a list of interactions and solution tactics. The authors do not further specify how this interaction list is then processed by a tool or which solution tactics are generated. They also do not provide interaction types to specify execution ordering (e.g., there is no precedes interaction).

In the context of data dependencies, various works exist in the area of concurrent data management. In one of the earliest, Bernstein [2] gives hints on when the order of instructions modifying or reading data matters: Three conditions are identified, namely flow dependence, anti-dependence (mirrored flow dependence),

and output dependence, which have all influenced our data dependency considerations of Table 1. Bernstein et al. [3] have shaped terms related to database management systems such as serializability and recoverability. However, this and related works are mainly about interleaving multiple concurrent transactions in a serializable manner at runtime whereas in our approach, we aim at ordering data-accessing actions at design time.

5.2 AOP Approaches

Shaker and Peters [17] use UML class and statechart diagrams for modeling core concerns (functional) and aspects (non-functional). In addition, they provide a statechart weaving language for weaving the functional and non-functional behavior. The authors present a static analysis of their design model which produces a list of potential core-aspect and aspect-aspect interactions. The verification is done on the woven model. The authors focus more on the static analysis and verification of their design model whereas in our approach we also provide resolution strategies and the guided-modeling procedure.

Nagy et al. [11] analyze problems and requirements for the composition of multiple aspects that match at the same join point. Furthermore, they propose a general declarative model for defining constraints upon possible aspect compositions and show how this model can be applied to AspectJ [10] and Compose* [6]. The main requirements they identified are that it should be possible to specify the execution order of aspects and to define conditional dependencies. There are three types of constraints, namely the *pre* (an aspect has to be executed before another), the *cond* (an aspect is executed depending on the outcome of another aspect), and the *skip* constraint (an aspect execution might be skipped). The constraints are then used to generate a set of concrete valid execution orders. The problem with this strategy is that the set of valid orders strongly depends on the quality and quantity of interdependencies. The lower the quantity of interdependency information, the higher the number of possible execution orders being generated. To overcome this complexity, our approach is rather supportive and still involves human interaction. This is comparable to business process modeling which is also done by human experts. However, in our approach the interdependencies can also be enriched by the use of data impact properties.

Katz [9] describes different categories (impact types) of aspects in terms of semantic transformations of state graphs of the base systems: spectative (read-only), regulative, and invasive (modify) ones. He defines syntactical identification procedures to determine the category of an aspect. The main goal of introducing categories is to simplify proofs of correctness, e.g., w.r.t. liveness and safety properties. Our approach also aims for the correctness, but mainly on the correctness of the composition of aspects (NFAs) and not on how aspects influence the correctness of the base system. Furthermore, in our approach, the category is determined manually as all aspects are considered black boxes whereas Katz uses statical analyses to automatically determine the category of an aspect.

Durr et al. [7,8] abstract the behavior of advices into a resource operation model which presents common or shared interactions. This model is complemented by a conflict model including data and control flow conflicts. The impact of advices on data is classified as read, nondestructive write (add), destructive write (modify), and unknown. They also identify conflicting situations with pairs of those impact types. In their analyzing process, a message flow graph is generated which represents all possible paths through the filter set (their approach is based on Composition Filters). This graph is then used to find the conflicting paths. In our approach, we also use data or control conflicts (impact types), but we use this information to enrich our interdependency model which is on a higher level than the concrete execution paths. Moreover, the constraints on the composition implied by the interdependencies are not only used for validation purposes but also for conflict resolution and guided composition. Another difference is that Durr et al. do static code analysis to determine impact types of advices whereas in our approach this is done manually.

6 Conclusions

In this paper, we presented our interdependency model which can be used to discover conflicts in compositions and non-functional behavior already at design time in order to avoid conflicts at runtime. Additionally, action properties have been introduced. The benefit of these properties is that they allow — in contrast to interdependencies — to regard one action in isolation. This helps to discover additional interdependencies even across different non-functional domains. Our rich interdependency model also enables conflict resolution and supports a guided, conflict-free composition procedure.

The presented concepts have been evaluated in the context of web services. A realistic web service example has been introduced in order to show the applicability and feasibility of our approach. Moreover, a set of graphical Eclipse-based editors has been implemented in order to support modelers during the complex task of action composition. Finally, related work has been identified showing that most approaches rely on completely automated processes, do not involve human actors, and thus strongly depend on the quality of the available interdependency information. These drawbacks are addressed in our approach by firstly obtaining a rich interdependency model with help of data dependencies and secondly by introducing our guided composition procedure.

However, our approach also has a few limitations. First, loops in the functional or non-functional composition logic are not yet supported. Second, interdependencies between functional and non-functional actions are currently out of scope. Third, conditions for process branching are not processed by our validation algorithm. These issues need to be investigated in the future.

Acknowledgements. This work has been partially supported by the German Federal Ministry of Education and Research (BMBF) in the project InDiNet (01IC10S04A).

References

1. Beauvois, M.: Brenda: Towards a Composition Framework for Non-orthogonal Non-functional Properties. In: Distributed Applications and Interoperable Systems (DAIS 2003), Paris, France, pp. 29–40 (November 2003)
2. Bernstein, A.J.: Analysis of programs for parallel processing. IEEE Transactions on Electronic Computers EC-15(5), 757–763 (1966)
3. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency control and recovery in database systems. Addison-Wesley Longman Publishing Co., Inc., Boston (1987)
4. Bowen, T.F., Dworack, F.S., Chow, C.H., Griffeth, N., Herman, G.E., Lin, Y.-J.: The feature interaction problem in telecommunications systems. In: Seventh International Conference on Software Engineering for Telecommunication Switching Systems, SETSS 1989, pp. 59–62 (July 1989)
5. Clark, J., DeRose, S.: XML Path Language (XPath) Version 1.0 (November 1999)
6. de Roo, A.J., Hendriks, M.F.H., Havinga, W.K., Durr, P.E.A., Bergmans, L.M.J.: Compose*: a language- and platform-independent aspect compiler for composition filters. In: First International Workshop on Advanced Software Development Tools and Techniques, WASDeTT 2008, Paphos, Cyprus (July 2008)
7. Durr, P., Bergmans, L., Aksit, M.: Reasoning about semantic conflicts between aspects. In EIWAS 2005: The 2nd European Interactive Workshop on Aspects in Software, Brussels, Belgium, pp. 10–18 (September 2006)
8. Durr, P., Bergmans, L., Aksit, M.: Static and Dynamic Detection of Behavioral Conflicts Between Aspects. In: Sokolsky, O., Taşıran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 38–50. Springer, Heidelberg (2007)
9. Katz, S.: Aspect Categories and Classes of Temporal Properties. In: Rashid, A., Aksit, M. (eds.) Transactions on Aspect-Oriented Software Development I. LNCS, vol. 3880, pp. 106–134. Springer, Heidelberg (2006)
10. Kiczales, G., Lamping, J., Mendhekar, A., et al.: Aspect-oriented Programming. In: Aksit, M., Auletta, V. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
11. Nagy, I., Bergmans, L., Aksit, M.: Composing Aspects at Shared Join Points. In: Hirschfeld, R., Kowalczyk, R., Polze, A., Weske, M. (eds.) NetObjectDays (NODe/GSEM), Erfurt, Germany. LNI, vol. 69, pp. 19–38. GI (September 2005)
12. OMG. Business Process Model and Notation (BPMN) 2.0 (January 2011)
13. Pulvermüller, E., Speck, A., Coplien, J.O., D'Hondt, M., De Meuter, W.: Feature Interaction in Composed Systems. In: Frohner, A. (ed.) ECOOP 2001. LNCS, vol. 2323, pp. 86–97. Springer, Heidelberg (2002)
14. Sanen, F., Truyen, E., Joosen, W.: Managing Concern Interactions in Middleware. In: Indulska, J., Raymond, K. (eds.) DAIS 2007. LNCS, vol. 4531, pp. 267–283. Springer, Heidelberg (2007)
15. Schmeling, B., Charfi, A., Mezini, M.: Composing Non-Functional Concerns in Composite Web Services. In: IEEE International Conference on Web Services (ICWS 2011), Washington DC, USA, pp. 331–338 (July 2011)
16. Schmeling, B., Charfi, A., Thome, R., Mezini, M.: Composing Non-Functional Concerns in Web Services. In: The 9th European Conference on Web Services (ECOWS 2011), Lugano, Switzerland, pp. 73–80 (September 2011)
17. Shaker, P., Peters, D.K.: Design-level detection of interactions in aspect-oriented systems. In: Proceedings of the 20th European Conference on Object-Oriented Programming, Nantes, France, pp. 23–32 (July 2006)