

Adaptive Approach for Impact Analysis in Enterprise Architectures

Melanie Langermeier^(✉), Christian Saad, and Bernhard Bauer

University of Augsburg, Augsburg, Germany
{[langemeier](mailto:langemeier@ds-lab.org),[saad](mailto:saad@ds-lab.org),[bauer](mailto:bauer@ds-lab.org)}@ds-lab.org

Abstract. Due to the large size of typical Enterprise Architecture models, it is often difficult for humans to fully grasp their contents. Moreover, because of their inherent complexity, the task of generating additional value from these models is very challenging without suitable analysis methods. Impact analysis, which determines the effects of changes or failures on other architectural elements, can therefore provide valuable information for enterprise architects. Whether an element is affected depends on its context, i.e. its (transitive) connections to other elements and their semantics with respect to the analysis. In this paper, we propose a generic, context-sensitive approach to the implementation of impact analyses. This method relies on the technique of data-flow analysis to propagate the effects through the model. Since the analysis specification only relies on a set of relationship classes, it can be easily adapted to the needs of organization-specific EA meta models by providing custom mappings for the respective types.

Keywords: Enterprise architecture analysis · Impact analysis · Change propagation · Data-flow analysis

1 Introduction

Enterprise Architecture Management (EAM) provides methods for managing the inherent complexity of the large IT infrastructures encountered in many organizations. Since the introduction of EAM is a complex and time consuming task, it is crucial that benefits are visible in early phases to enhance the acceptance of the initiative. Therefore, methods are required that allow all involved organization units to make use of the gathered data. Enterprise Architecture (EA) models usually contain many elements, which are connected through complex relationships. (Semi-)automatic analysis techniques facilitate gaining benefits in early stages as well as leveraging the models once EAM has been successfully introduced.

Although much research has been carried out in the EA domain, most of this work focuses on methodologies for the development and the representation of enterprise models rather than on techniques which explore possible applications scenarios [1,2]. Analysis of EA models is mostly limited to quantification approaches, which encompass the definition and computation of quality

attributes such as application usage and service availability [1]. Further proposals include the evaluation of performance and cost aspects in the different layers of enterprise models [3] and a catalog of KPIs for measuring EA management goals [4]. Because the set of classes, properties and relationships often evolves during the EAM lifecycle while these methods typically rely on a fixed language structure, their applicability can be limited, especially in early phases of an EA initiative.

The second main category of analysis approaches found in canonical literature consists of the so-called impact analysis. These methods simulate the effects of a change (e.g. modification of a CRM system) or a failure (e.g. shut-down of a server) to assess risks in the current architecture [5]. The information is generated by an evaluation of the dependencies between the architecture's constituents. To make proper assertions about these relationships, it is vital to examine each element in its respective overall context, meaning that relationships with other elements in the model have to be taken into consideration. For example, to examine the impact of a server failure on business processes, it must be determined which applications rely on this server. This requires a careful evaluation of indirect and transitive paths to ensure that all necessary information is retrieved while, at the same time, excluding irrelevant relationships. As mentioned before, existing approaches and tools for designing and analyzing EA models usually rely on a static meta model structure. This poses a challenge as organizations tend to employ customized languages, making the adaption of existing analyses very difficult [6]. To address this challenge, more flexible methods for handling structural dependencies are required.

In this paper, we present a robust technique for analyzing impacts in EA models. It relies on the principle of data-flow analysis, a method which originates from the field of compiler construction, to propagate contextual information along the model's edges. We combine this method with a classification of relationship types that enables an easy adaption to different conventions by providing mappings for the relevant elements. It is also possible to extend the proposed definitions with individual impact propagation rules. To demonstrate the viability as well as the generic applicability of this approach, we implement multiple impact analyses for different EAM languages.

2 Impact and Dependency Analysis

According to [7], determining the effects of a change requires an iterative and discovery-based approach. Change impact analysis can be performed for a single software system, but also on an architectural level for a full application landscape or an enterprise architecture. A related topic which is also of interest in this context is the analysis of dependency relationships. Changing a model element typically causes further changes at its neighboring elements (*direct impact*). However, as these changes may in turn affect other elements (*indirect impact*), the effect propagates throughout the model. Consequently, even a small change in a single element can cause ripple-effects, resulting in non-trivial consequences.

While the direct impact can be derived from the connectivity graph, the computation of indirect impacts (*n-level impacts*) requires reachability information. However, since this method approximates potential impacts, it tends to overestimate the result by generating false-positives. The precision of the analysis can be improved by using a constraint mechanism or by incorporating structural and semantic information [7].

Most of the work regarding impact analysis of software focuses on the code level [8]. Approaches which evaluate architectures usually only regard concepts such as components, packages, classes, interfaces and methods. Due to the limited amount of supported types and lower complexity regarding the relationships between those, these approaches are not suitable for use in EAM. For example organizational aspects cannot be considered. Nevertheless, some techniques which target the UML are more closely related to the EAM domain. Reference [9] propose a methodology for subjecting analysis and design documents to an impact analysis to detect side effects of changes in the context of UML-based development. To restrict the set of affected model elements they propose the use of a coupling measure and a predictive statistical model. The impact analysis itself is specified using the OCL. Nevertheless the implementation is restricted to UML-based documentation, other modeling languages like ArchiMate or self-developed DSLs are not treated. Reference [10] developed an approach which supports traceability by providing requirements engineers, project planners and maintainers with the ability to monitor the effects that changes have on software systems. They differentiate between three types of relationships to define the traces: *representation*, *refinement* and *dependency*. To determine the change impact, they (semi-)automatically analyze requirement traces using these three categories.

References [6, 11] propose techniques for EA dependency analysis. Saat focuses on time-related aspects (org. “zeitbezogene Abhängigkeitsanalysen”) by considering for each element its life time, the status (current or proposed) as well as the life cycle phase with its duration. However, no execution or implementation details are provided for this approach. Kurpjuweit and Aier developed a formal method for flexible and generic dependency analysis. To determine dependent elements, they use the transitive closure of a set of relations. They also define an expansion function, which allows to consider special relation semantics, e.g. hierarchical refinement or reflective relation types.

Reference [12] as well as [13] propose the use of Bayesian Belief Networks (BBN) for EA modeling. These approaches rely on causal dependencies as well as inference methods for BBN and a diagnosis analysis to determine the impact. The former realizes a failure impact analysis, theoretically described in the pattern catalogue [14], using the diagnostic analysis [15] and the modeling tool GeNIe. As a result, architectural components can be ranked with respect to their criticality for a business process. However, this approach focuses on availability, not on changes. Tang et al. employ a combination of predictive reasoning to determine affected elements and diagnostic reasoning to determine the cause of a change. Prior to the analysis, the architect has to assign a probability to each root node and a conditional probability table to each non-root node.

Propagation rules are another method for determining the impact of changes. This technique allows to define effects that depend on structural and semantical properties. An iterative application of those rules to a model yields the direct and indirect impacts. Reference [5] present such rules for the most important relationships in ArchiMate models, differentiating between the *removal*, the *extension* and the *modification* of an architectural element. However, the definitions are given in an informal and textual manner and no technical realization is supplied [16] propose rules that encode the dependency relationships of the attributes of entities. Changes are thereby propagated to determine the impact on a defined set of element types, namely business goals, processes, services and infrastructure components as well as the relations *runs on*, *provides*, *executes* and *delivers*. No mechanism is specified for implementing the change propagation. Reference [17] also rely on the propagation concept to define a conceptual coupling measurement for software components. Based on this information a dependency matrix is established which allows to predict change impacts. In [18], a tool for impact-of-change analysis is described. The author represents enterprise architectures in XML and uses the Rule Markup Language (RML) to define transformations which represent the rules which define the impact-of-change. The RML rules are analyzed through a pattern matching of the antecedent against the input XML. If a rule matches, the variables will be bound and an output XML is generated based on the rule output.

3 Foundations

By definition, the assessment of an impacts has to include the computation of reachable elements. According to [7], *dependability* refers to directly connected elements while *reachability* additionally regards transitive connections. To evaluate these relationships, we employ data-flow analysis, a technique based on the principle of information propagation which allows for declarative and recursive specifications. Consequently, we can directly implement the following definition: *An element is reachable if at least one predecessor element is reachable*. In this context, a predecessor is defined as the source element of an incoming edge. Since there are typically no isolated areas in an EA model, this would however result in all elements being classified as *reachable*. For a more focused analysis, we therefore need to extend this specification with contextual information. For this purpose, we establish a categorization mechanisms for relationships (see Sect. 4.1). A set of propagation rules defined for each relationship class then indicates how changes will be propagated to neighboring elements.

In the following, we introduce a generic representation of model and meta model data, which will ensure the applicability of the defined analyses in the case where organizations employ customized versions of the underlying EA meta model. Afterwards we exemplify the use of data-flow based specifications by implementing a naive reachability analysis, which will subsequently be extended to enable a context-sensitive analysis of impacts.

3.1 Formalizing the Meta Model and the Model

The high diversity of meta models in the EAM domain poses a major challenge to any technique in this area. To overcome this issue, we employ a generic meta model with the ability to support any EA language based on traditional modeling paradigms. By combining both meta model and model data in a single representation, this approach abstracts from the particular structure of a given input language and allows for generalized analysis specifications.

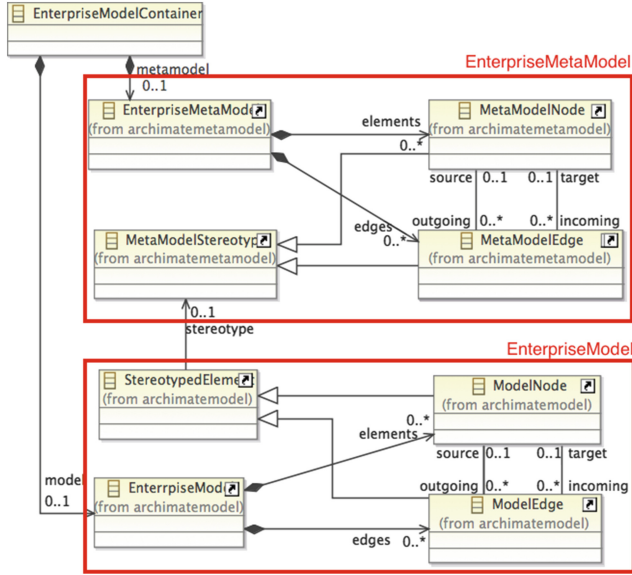


Fig. 1. Generic representation for EA (meta) model data

A condensed version of this specification is depicted in Fig. 1. The relevant elements can be described as follows: Each concept of the EA language is translated into either a *MetaModelNode* or a *MetaModelEdge*. Connections between these elements can be established dynamically during the transformation process, based on their actual usage in the EA model. Both types also carry additional meta information such as their stereotype, the concept's name and its properties. Elements from the EA model are converted into *ModelNodes* and *ModelEdges*. Their type is represented as a stereotype relationship to the respective *MetaModelNode* or *MetaModelEdge*.

3.2 Analyzing Reachability for Enterprise Architecture Models

The computation of reachability information forms the basis for a subsequent impact analysis. An element is declared reachable, if there exists a path connecting the element to the starting point (*indirectly connected elements*). The

reachability analysis is carried out using the Model Analysis Framework (MAF) [19] which supports the specification and execution of data-flow based model analyses.

Data-flow analysis is typically used by compilers to derive optimizations from the structural composition of program instructions. Canonical examples include the calculation of reaching definitions and variable liveness. For this purpose, the program is converted into a control-flow graph with the nodes representing the basic blocks and the edges denoting the flow of control. A set of data-flow equations is then evaluated in the context of each node. Each equation takes the results computed at the immediate predecessor nodes as input, applies a confluence operator (union or intersection) to combine these sets and finally modifies the values to reflect the effects of the local node's instructions. Effectively, this method describes an equation system which propagates information throughout the underlying graph, thus enabling a context-sensitive evaluation of each instruction. If loops are present, fixed-point evaluation semantics are employed to approximate the runtime behavior of the program.

In [20] we discussed an adaption of this analysis technique to the modeling domain which we referred to as a *generic “programming language” for context-sensitive model analysis*. This approach defines a declarative specification language that allows the annotation of data-flow attributes at meta model classes that can subsequently be instantiated and evaluated for arbitrary models. This technique has several significant advantages: Data-flow analysis provides inherent support for the implementation of recursive specifications which iteratively propagate information through a model. Also, since information is routed along model edges, each model element can be evaluated in its overall context, thus eliminating the need for static navigational expressions which are common in languages such as OCL. This is important in the EAM domain where the structure of both meta models and models is highly dynamic. Finally, the usage of fixed-point semantics allows to implement a correct handling of cyclic paths. Using MAF, a reachability analysis for model elements can be specified in the following way:

```

1: analysis reachability_analysis {
2:   attribute is_reachable : Boolean initWith false;
3:   extend node with {
4:     occurrenceOf is_reachable calculateWith
5:       self.incoming.source.is_reachable()
6:       ->includes(true);
7:   }
8:   extend startnode with {
9:     occurrenceOf is_reachable calculateWith true;
10:  }
11: }
```

As described above, an element $e1$ is reachable from another element $e2$, if there exists a path between $e1$ and $e2$. Here, we assume that the meta model

defines the classes *node* and *startnode*, the latter being a specialization of the former one. We further classify changed elements in the model as *startnodes* for the analysis. The reachability status is computed by a data-flow attribute *is_reachable* of type *boolean* which is initialized with the value *false* (line 2). Lines 3-7 attach this attribute to all instances of the *node* class. To determine the reachability status of a node, the data-flow equation in lines 5-6 accesses the *is_reachable* values computed at the respective node's predecessors, thereby directly implementing the recursive specification. Finally, lines 8-10 overwrite this equation at *startnodes* which are, by definition, always reachable.

By combining the reachability analysis with the generic meta model approach from Sect. 3.1 enterprise architecture models can be analyzed independently from their meta model structure. However, at this point the analysis will yield mostly false positives regarding change or failure impacts as it does not consider language semantics.

4 Analyzing the Impact

We will now extend the reachability analysis principle with context-specific declarations to restrict results to a meaningful subset of elements and to reflect different types of changes. For this purpose, we employ a class configuration approach similar to [10]. This decision followed a failed attempt to reuse the classes identified by von Knethen et al. for which we could not determine sound mappings for the ArchiMate language. The proposed classes are based on a literature review and are presented in Sect. 4.1. While the classification approach ensures the general applicability of the method, the implementation of the context-dependent impact analysis is based on a set of rules which differentiate between the class semantics as well as change and relationship types to propagate the correct impact information through the model.

It is important to note that, due to the lack of detailed information in enterprise architecture models, an accurate assessment of impacts is not possible. We decided against the use of probabilistic models because of the inherent uncertainty when defining the thresholds (although, if desired, the technique could be extended to compute probabilities for each effect). Instead, we approximate impacts through a best case/worst case analysis similar to the practices in software analysis. The worst case represents the maximal set of affected elements, whereas the best case conforms to the minimal set. The actual impact (which must be determined by a domain expert) typically lies somewhere in between. Furthermore, the proposed analyses can quantify specific changes.

In the next sections, we present our classification approach for EA relationships followed by case studies that implement different types of analyses: The prediction of change impacts is exemplified in Sect. 4.2 while Sect. 4.3 addresses failure impacts. Finally, Sect. 4.4 demonstrates impact quantification.

4.1 Relationship Categorization

Based on a literature review of existing EA frameworks and their meta models, we classified the relationship types of enterprise architectures according to their semantics. This includes the Core Concepts Model (CC) of ArchiMate [21] and the DM2 Conceptual Data Model of DoDAF [22].

Overall, we were able to identify five classes of relevant EA relationship types: *Located at* denotes the allocation to some location or organization unit. Any kind of provisioning of functionality, information or behavior is of the type *provide* while the *consume* class denotes the consumption of those elements. *Structural dependent on* relationships define the structure or organization of entities in a single layer. The *behavioral dependent on* class, on the other hand, summarizes relationships which declare dependencies between the behavior of elements in a single layer which are neither of the type *provide* nor *consume*. It should be noted that a relationship can belong to multiple categories. In this case the strongest rule (i.e. the rule with the greatest impact) will be chosen for the worst case analysis while the best case analysis uses the weakest one (resulting in the lowest impact). Table 1 lists all classes along with corresponding examples from the ArchiMate Core Concepts and the DoDAF DM2. Note that the mapping in Table 1 is a suggestion based on our interpretation of the concepts and can be adapted if an organization assigns different semantics to these types.

Table 1. Classification of EA relationships

Relationship class	CC ArchiMate	DM2 DoDAF
located at	assigned to	is-at
provides	realizes, accesses	provides, performs
consumes	uses, accesses	consumes
structural dependent on	aggregated by, composed by	part-of
behavioral dependent on	triggered by, flow from	

To formalize the change semantics of the relationship classes, we employ the following syntax:

$$A.X \rightarrow B.Y$$

This statement indicates that, if element *A* has the characteristic *X*, then element *B* will have the characteristic *Y*. *A* and *B* are the source and the target of the relationship while *X* and *Y* represent specific impact characteristics. For the change impact analysis (see Sect. 4.2), this results in $X, Y \in \{no\ change, extend, modify, delete\}$ while for the failure impact analysis (see Sect. 4.3), the possible values are $X, Y \in \{available, not\ available\}$. Change operations can be clustered on the left hand side: $A.\{X, Y\} \rightarrow B.Z$ states that, if *A* has characteristic *X* or *Y*, *B* will have the characteristic *Z*. On the right hand side of the rule, one

can optionally differentiate between a worst case (WC) and a best case (BC) impact. It is generally possible that changes in enterprise architectures affect the relationships themselves. In the current state of the analysis specifications, the impact of a deletion or insertion of a relationship is ignored.

4.2 Propagation Rules for Change Impact

The change impact analysis differentiates between the three change types *extend*, *modify* and *delete* as proposed by [5]. Extensions refer to cases where new issues are added but the initial functionality or structure remains the same. Consequently, extensions do not propagate to depending elements. By contrast, a modification also affects the functionality or the structure and therefore it cannot be guaranteed that initially provided issues will still be available or that their behavior remains unchanged. Finally, deletion indicates that an element will be removed from the enterprise architecture. The deletion of an element may trigger additional changes to the architecture. For example, if an organization unit is removed, hosted application components must be assigned to another host. This impact would be interpreted as an *extension* of the affected element, in this case the application component.

The change types are prioritized as follows: *delete* overrides *modifies* overrides *extends* overrides *no change*. Depending on the requirements, additional change types can be implemented. Change rules for the relationship classes are shown in Table 2. Assuming, for example, that an application component A realizes a

Table 2. Impact rules for the relationship classes

Class	Rule
located at	A.{del,mod,ext} → B.NO
	B.del → WC: A.del BC: A.ext
	B.{ext,mod} → WC: A.mod BC: A.NO
provides	A.del → WC: B.del BC: B.ext
	A.mod → WC: B.mod BC: B.NO
	A.ext → WC: B.ext BC: B.NO
	B.{del,mod,ext} → A.NO
consumes	A.{del,mod,ext} → B.NO
	B.{del,mod} → WC: A.mod BC: A.ext
	B.ext → A.NO
structurally dependent on	A.del → WC: B.del BC: B.mod
	A.{mod,ext} → B.NO
	B.{del,mod} → WC: A.mod BC: A.NO
	B.ext → WC: A.ext BC: A.NO
behaviorally dependent on	A.{del,mod,ext} → B.NO
	B.{del,mod,ext} → A.NO

service B , this connection will be mapped to the class *provides*. The rules indicate that, if the component is deleted, the service has to be deleted, too (worst case). In the best case scenario, another application will implement the service in which case the rule $A.del \rightarrow WC: B.del \quad BC: B.ext$ applies.

A modification of the component could also necessitate a modification of the service as support for required functionality is no longer available. For the best case, we assume that the modification will not affect critical service functions which is expressed by the rule $A.mod \rightarrow WC: B.mod \quad BC: B.NO$. Finally, an extension of the application component is handled similarly to the modification scenario: In the worst case, the service must also be extended while, in the best case, the service is unaffected ($A.ext \rightarrow WC: B.ext \quad BC: B.NO$). Conversely, if the service is changed, this has no effect on the realizing application component as expressed by $B.\{del, mod, ext\} \rightarrow A.NO$.

In addition to the classification along the lines of their EA semantics, a differentiation between impact types can be useful as well. We therefore define the following three effects: *strong*, *weak* and *no effect*. The type of effect must be specified for each direction of a relationship. The notation $X - Y$ indicates that a change in the source has an effect of type X on the target and, vice versa, a change in the target has the effect of type Y on the source. This leads to six effect classes: *Strong-Strong*, *Strong-Weak*, *Strong-No effect*, *Weak-Weak*, *Weak-No effect* and *No effect-No effect*. The semantics of the effects are defined using the rule schema introduced in Sect. 4.1 and are depicted in Table 3.

Table 3. Impact rules for the effect classes

Effect	Rule
strong	$A.del \rightarrow WC: B.del, BC: B.ext$
	$A.mod \rightarrow B.mod$
	$A.ext \rightarrow B.ext$
weak	$A.del \rightarrow WC: B.mod, BC: B.no$
	$A.mod \rightarrow WC: B.mod, BC: B.ext$
	$A.ext \rightarrow WC: B.ext, BC: B.NO$
no effect	$A.\{del, mod, ext\} \rightarrow B.NO$

If A strongly affects B , this indicates that, if A is deleted, B either has to be deleted as well (worst case) or must be extended (best case). A modification of A leads to a modification of B and the same applies to extensions. If, for example, an application component realizes a service, then the application component has a strong impact on the service while the service may only have a weak impact on the application component. This specific interpretation of *realize* would result in an assignment to the *Strong-Weak* class. A weak effect denotes that the deletion of A conducts no change in B in the best case, and a modification in the worst case. A modification of A in the worst case requires a modification of B . In

Table 4. Mapping of ArchiMate and DoDAF concept to the effect type classes

Effect type class	CC ArchiMate	DM2 DoDAF
Strong-Weak	realizes	provides, performed by
Weak-Weak	triggered by, flow from	
No effect-Strong	aggregated by	part-of
No effect-Weak	uses, assigned to	consumes

the best case it has only to be extended. Finally, if A is extended, in the best case B must not be changed, in the worst case it has to be extended, too. If the relationship is mapped to *no effect*, any change of A has no effect on B. A possible mapping of ArchiMate relationships and DoDAF relationships to the effect type classes is shown in Table 4.

4.3 Propagation Rules for Failure Impact

In the context of enterprise architectures, failure impact analysis can be used to simulate the effect of the non-availability of an architectural element. Examples for failures are the shut-down of a server or inaccessible applications. Potential effects could be a defunct business process or a non-available product. The different kinds of failures and effects are categorized as *available* (AV) and *non-available* (NON). Non-available means that the respective element does not exist or cannot provide its functionality. The failure impact rules for the relationship classes are defined in Table 5. We again exemplify the rules using the class *provides* for an application component (A) that realizes a service (B). If the application component is down, i.e. not available, the realizing service is also classified as not available: $A.NON \rightarrow B.NON$. However, if the application is up and running, we cannot draw conclusions on the availability of the service (expressed by $A.AV \rightarrow \emptyset.\emptyset$). Conversely, the availability status of the service has no effect on the application: $B.\{AV, NON\} \rightarrow \emptyset$. Table 5 shows three different effect types for the failure impact analysis:

No effect will be propagated ($A.NON \rightarrow \emptyset$).

Potential Impact: In the worst case, the connected element also will be no longer available. In the best case, no effect will be propagated ($A.NON \rightarrow WC : B.NON$; $BC : \emptyset$).

Impact: A failure always causes a failure of the connected element ($A.NON \rightarrow B.NON$).

Since as long as an element is available ($A.AV$), no information is propagated, we do not consider this case for the three effect types. Similar to the change impact analysis, we propose effect classes as an alternative to the relationship classes. A possible mapping of the effect classes to concepts from ArchiMate and DoDAF is shown in Table 6. In this case, the effect class *No change - Potential impact* is mapped to the ArchiMate type *uses*. This means that, if service A uses

Table 5. Failure impact rules for the relationship classes

Class	Rule
located at	$A.\{AV, NON\} \rightarrow \emptyset$
	$B.AV \rightarrow \emptyset$
	$B.NON \rightarrow WC: A.NON \ BC: \emptyset$
provides	$A.AV \rightarrow \emptyset$
	$B.\{AV, NON\} \rightarrow \emptyset$
	$A.NON \rightarrow B.NON$
consumes	$A.\{AV, NON\} \rightarrow \emptyset$
	$B.AV \rightarrow \emptyset$
	$B.NON \rightarrow WC: A.NON \ BC: \emptyset$
structurally dependent on	$A.AV \rightarrow \emptyset$
	$A.NON \rightarrow WC: B.NON \ BC: \emptyset$
	$B.AV \rightarrow \emptyset$
	$B.NON \rightarrow WC: A.NON \ BC: \emptyset$
behaviorally dependent on	$A.AV \rightarrow \emptyset$
	$A.NON \rightarrow WC: B.NON \ BC: \emptyset$
	$B.AV \rightarrow \emptyset$
	$B.NON \rightarrow WC: A.NON \ BC: \emptyset$

Table 6. Mapping of ArchiMate and DoDAF concept to the change effect type classes

Effect type class	CC ArchiMate	DM2 DoDAF
No change - Potential impact	assigned to, uses	consumes
Potential impact - Potential impact	aggregated by, triggered by, flow from	part-of
Impact - No change	realizes	provides, performed by

application B and the service is no longer available, the application will not be affected. On the other hand, if the application is no longer available, this could have an impact on the availability of the service (worst case).

4.4 Quantification of the Impact Analysis

It is possible to extend the rule definitions with the ability to quantify a change (e.g. in terms of costs) by implementing additional data-flow attributes. For example, to compute potential savings on IT maintenance, the maintenance costs of all deleted applications and infrastructure components and their corresponding services could be aggregated.

A further customization consists of a modification of the rule set to support change probabilities. Instead of computing a single status, we could define four separate data-flow attributes, which compute the respective probabilities for the types *delete*, *modification*, *extension* and *no change*. This would also require an extension of the rule specifications: A rule in the format $P(A.del) = X \rightarrow P(B.del) = 0.8 \cdot X$ indicates that, if the probability that A is deleted is X, then the probability that B has to be deleted is $0.8 \cdot X$ or, in other words, if A is deleted then in 80 % of the cases B will be deleted as well.

The general schema for a quantifying rule describing architectural elements A/B, properties K/L and $X/Y \in \mathbb{R}$ is as follows: $P(A.K) = X \rightarrow P(B.L) = Y$. This rule indicates that, if the property K of A has the value X, then the property L of the element B has the value Y. P is a function that assigns a value (e.g. probability or costs) to the property of the element. Instead of supplying a concrete value for Y, it is also possible to use a mathematical operation such as $P(B.L) = X + X \cdot Y$

Quantifying the Change Impact Analysis. To introduce a quantification of the change impact analysis, probabilities can be assigned for extension, modification or deletion, meaning that every architectural element has a probability value for $P(A.ext)$, $P(A.mod)$, $P(A.del)$ as well as for $P(A.NO)$, i.e. when no change has to be made. The organization-specific propagation rules can be defined based on experience or an analysis of past change effects. Because of the high abstraction level of early EA designs, the probabilities can be roughly estimated. The rule for a strong effect in the case of deletion could look as follows:

$$\begin{aligned} P(A.del) &= X \rightarrow P(B.del) = 0,5 \cdot X \\ P(B.ext) &= 0,25 \cdot X \\ P(B.NO) &= 0,25 \cdot X \end{aligned}$$

If A is deleted, in 50 % of the cases B will be deleted to, in 25 % of the cases B must only be extended or won't be subject to any change. The probability for a modification is 0 %. Note that the sum of $P(B.del)$, $P(B.ext)$, $P(B.mod)$ and $P(B.NO)$ is 100 %. In practice, it is vital that analysis results are compared to the actual outcome in order to refine the probability values of the propagation rules.

Quantifying the Failure Impact Analysis. In [1], the authors determine the availability of IT services by considering the application and infrastructure layer. Based on this research, we present a quantification of the failure impact analysis using availability measures. The average availability is defined as the repair rate of the component divided by the sum of the repair and failure rate. This value represents the probability that a service is available to its users. The original implementation utilizes Fault Tree Analysis (FTA) by introducing gates that indicate AND or OR dependencies between architectural elements under the assumption of independence of failures, passive redundancy, perfect switching and no repairs. For the AND case, all connected elements must be available while in the OR case, a single available element is sufficient. To adapt

Table 7. Mapping of ArchiMate and DoDAF concept to the failure effect type classes

Effect type class	Relationship category
OR - NONE	provides (BC), structurally dependent on (BC)
NONE - OR	consumes (BC), behaviorally dependent on (BC)
NONE - AND	located at (BC, WC), consumes (WC), behaviorally dependent on (WC)
AND - NONE	provides (WC), structurally dependent on (WC)

this analysis to our method, we mapped the relationship classes identified in Sect. 4.1 to the AND/OR cases, depending on the best and worst case scenarios.

Table 7 shows the mapping of the relationship classes to the propagation semantics. With source element A and target element B of a relationship, OR-NONE for *provides* (BC) is interpreted as follows: In the best case, B is available if at least one of the providers (A) is available (the availability of B has no effect on the availability of A). In the worst case, the rule AND - NONE applies which demands that, for B to be available, all providing elements (A) must be available. There are two alternatives for the specification of the failure impact analysis rules, which are presented below.

Alternative 1 for Failure Impact Analysis. Availability AV is defined by the parts AV_L , AV_P , AV_C , AV_S and AV_B while the total availability conforms to $AV := AV_L \cdot AV_P \cdot AV_C \cdot AV_S \cdot AV_B$. To determine the overall availability of an element, the result is computed dynamically according to the stated equation. The different parts are calculated using the rules from Table 8. Each part is defined by a single rule. The table exemplifies the computation for AV_P and relationships in the *provides* class:

Table 8. Availability propagation rules for quantification (composed availability)

Class	Rule
...	...
provides	$P(A.AV) = X \rightarrow WC: P(B.AV_P) = B.AV_P * X$ $BC: P(B.AV_P) = B.AV_P + X - (B.AV_P * X)$ $P(B.AV) = X \rightarrow P(A.AV_P) = N.A.$
...	...

The worst case scenario demands that all providing components are available. In the best case, only a single providing component is necessary. Conversely, the availability of B has no influence on the provider A.

Alternative 2 for Failure Impact Analysis. Instead of decomposing the availability property into different parts, it is also possible to introduce a universal quantifier for the propagation rules. In this case, the propagation of a value

is dependent on all rather than on a single connected element. Thereby only elements which are connected via the same relationship classes are considered. The rule for the *provide* class is shown in Table 9.

Table 9. Availability propagation rules for quantification (allquantor)

class	rule
...	...
provides	$\forall A_i (P(A_i.AV) = X_i) \rightarrow$ $P(B.AV) = WC: B.AV * \prod X_i$ $BC: B.AV * (1 - \prod (1 - X_i))$ $A_i: \text{Every } A_i, \text{ where exists } (A_i \text{ provides } B)$ $P(B.AV) = X \rightarrow P(A.AV) = N.A.$
...	...

The universal quantifier on the left hand side indicates that all elements A_i must be considered that are connected to B via a relationship of the type *provide*. Each element has a specific availability value which is accumulated by the mathematical definition on left hand side. In the worst case, it is a multiplication (AND-semantic) and, in the best case, the co-product (OR-semantic).

4.5 Realization of the Rules

The rules defined in Sects. 4.2, 4.3 and 4.4 can be realized as data-flow equations. First, the meta model and model data has to be converted to the generic representation presented in Sect. 3.1. The change impact analysis is initialized with the change status of the elements (data-flow attributes for unmodified elements are initialized with *no change*). For the failure impact analysis, unavailable elements must be marked while other elements are set to *unknown*. Similarly, the known availability probabilities must be set for the quantified failure impact analysis. Using fixed-point semantics, these values are then iteratively recomputed to propagate the change effects. The following data-flow rule (written in Java) calculates the best case result for the change impact analysis based on the presented effect types (see Sect. 4.2 and Table 3).

The status of the current *context* node depends on the status of the connected elements as well as the direction of the relationship. Therefore, to correctly determine the change status, all incoming (lines 2-18) and outgoing edges (lines 19-21) have to be processed. The status value which has been computed for a connected element is retrieved through an invocation of *getStatus()* (line 3). This instructs the data-flow solver to recursively compute and return the requested value. Based on the type of the incoming edges, it is then decided whether there is a *strong effect* (line 5), a *weak effect* (10) or no effect (15) on the target. The concrete type of the change is determined by evaluating the status of the edge's source element (lines 6-9, 11-14, 16-17). Finally, *computeStatus()* is invoked to

```

1: Object node_changestatus_bestcase(Node context){
2:   for (Edge incomingEdge : context.getIncomingEdges()){
3:     Status sourceNodeStatus = incomingEdge.source.getStatus()
4:     Status contextNodeStatus = context.getStatus()
5:     if (incomingEdge.effectClass == StrongEffectTarget)
6:       if (sourceNodeStatus == (DEL||EXT))
7:         return computeStatus(contextNodeStatus, EXT)
8:       else if (sourceNodeStatus == MOD)
9:         return computeStatus(contextNodeStatus, MOD)
10:    if (incomingEdge.effectClass == WeakEffectTarget)
11:      if (sourceNodeStatus == (DEL||EXT))
12:        return computeStatus(contextNodeStatus, NO)
13:      else if (sourceNodeStatus == MOD)
14:        return computeStatus(contextNodeStatus, EXT)
15:    if (incomingEdge.effectClass == NoEffectTarget)
16:      if (sourceNodeStatus == (DEL||MOD||EXT))
17:        return computeStatus(contextNodeStatus, NO)
18:  }
19:  for (Edge outgoingEdge : context.getOutgoingEdges()){
20:    ...
21:  }
22: }

```

assess the status of the local element. This method implements the prioritization relationships between the change types, e.g. by ensuring that a weak change like *no change* cannot override a stronger one like *delete*. The target nodes of outgoing edges are evaluated similarly (lines 19-21).

All rules have been implemented as DFA equations and integrated into a plugin for the MID Innovator [23]. To illustrate our approach, we applied the change impact analysis to the MIDWagen architecture, an example that ships with the tooling. It describes the IT landscape of a car rental organization with its actors, business services, business processes, application components and services as well as the required infrastructure components and services.

We now assume that the *Booking System*, which is responsible for payment transactions, has to be modified due to security issues. We further assume that this change will only affect the *Payment* service while the *Bonus Booking* service does not have to be adapted. The modification of the *Payment* application service (AS) also causes a change to the supporting *Return* process. Other elements such as the *Payment* business service (BS) or the *Renter* role are not affected. Figure 2 shows the relevant excerpt from the MIDWagen model. For this scenario, we employ the effect type classification described in Sect. 4.1. Since the model is given in the ArchiMate language, we are able to use the mappings listed in Table 4.

In Fig. 2, the resulting worst case change propagation path is indicated by thick grey arcs. Solid lines represent paths along which a change is forwarded, while dashed lines denote relationships which are considered but do not result in change propagation. The respective change value is indicated by the labels *MOD* for modification and *NO* for no change. Only the relevant elements are labeled although the whole model will be evaluated. The final impact set consists

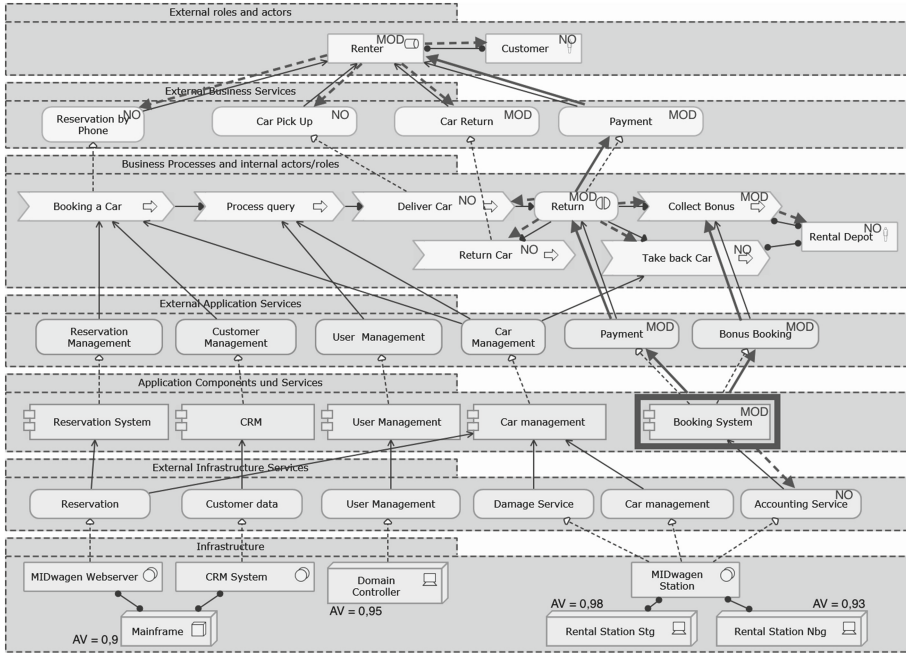


Fig. 2. Excerpt of the ArchiMate model for the MIDWagen use case [23] with the worst case change propagation path

of the elements $\{Payment\ AS\ modified, Bonus\ Booking\ modified, Return\ modified, Collect\ Bonus\ modified, Payment\ BS\ modified, Renter\ modified\}$, while the final result set for the best case analysis is empty. Both results represent realistic approximations which have to be interpreted considering the severity of the modification. In the best case scenario (e.g. performance issues), the modification of the *Booking System* does not affect the provided functionality, and therefore the service does not need to be changed. In the worst case, for example a substantial change in the functionality due to security issues, the effects of the change propagate to the business layer role which is potentially affected by the modification. Both impact sets represent approximations, which can be of great value for estimating the real effects of a change, especially in early design stages. The actual impact set of the change is $\{Payment\ AS\ modified, Return\ modified\}$ which lies in between the worst and the best case change set.

5 Evaluation

For evaluation purposes, the proposed methods have been implemented as a plugin for the enterprise architecture tool *Innovator* and applied the different kinds of analyses to the MIDWagen case study, which is shipped with the tool. Although it is not a real world example, the level of detail as well as the extensibility of the underlying EA language enabled a thorough evaluation of the

viability and the robustness of our technique. An excerpt of the MIDWagen case study and the application of the change impact analysis has been presented in Sect. 4.5. Overall, we evaluated the proposed approach in four different scenarios: The deletion of *User Management*, a modification of the *Booking System*, an extension of *Car Management* as well as an availability analysis with values set for infrastructure components. The first three scenarios were subjected to the change impact analysis (Sect. 4.2), the last one to the quantified failure impact analysis (Sect. 4.4). The expected impact was determined to lie between the worst and best case results although, in some cases, the gap between the both result types is quite large. For example, the worst case availability of the process *Take back car* is 29 %, while it is 91 % for the best case scenario. In this case, the mapping to the relationship classes did not properly reflect the semantics of the elements. We also observed that in some cases, mapping the available relationship types to the proposed relationship classes can be difficult. These experiences led to the establishment of the impact classifications for each analysis which enable a more detailed mapping of relationship types to the impact semantics. As a result, organization specific aspects can be represented in an improved fashion with the potential downside of a more time consuming mapping. It can be assumed that, by further refining the mappings, it is also possible to resolve the large differences between the worst and best case results.

By employing mappings along with a classification approach for relationships, the technique is universally applicable and, depending on the chosen classification technique, can rely on the semantics of the EA elements or focus on their impact semantics. In this paper, we provided mappings for the DoDAF DM2 as well as the ArchiMate core concepts (see Sects. 4.1, 4.2, 4.3 and 4.4). Especially at the beginning, the classification approach based on EA semantics was very useful due to the easier mappings although it tends to lead to less precise results. Conversely, the classification along the impact semantics required a higher specification effort with the benefit of an analysis which was better adapted to organization-specific needs and thus yielded better results.

Most of the research work regarding impact analysis has been carried out theoretically and thus has not yet been applied to real architecture models (e.g. [5, 6, 16]). An exception exists in the work of [13] who employ predictive and diagnostic reasoning in BBN. However, one disadvantage of their approach can be found in the high effort required to annotate probability information. The technique proposed in our paper simplifies analysis specification through a generic representation of model data and through predefined and extensible categorizations of relationships and effects. Many existing approaches do not address problems relating to cyclic dependencies or contradicting results, the latter one for example being a weakness of the tooling proposed by [18]. Furthermore, the issue of the scalability of the technique, which is based on pattern matching and model transformations is not considered and the employed RML technique is highly dependent on specific usage scenarios as well as on the respectively chosen EA language. By utilizing the data-flow analysis method with its inherent support for cyclic dependencies, recursive specifications and iterative

result computation, we are able to address these challenges. The scalability of DFA (and the Model Analysis Framework in particular) has been demonstrated in the context of other domains including the analysis of extensive AUTOSAR models [24].

6 Conclusion

In this paper we proposed a context-sensitive impact analysis technique for EA models. The approach relies on two underlying concepts: The problem of diverse EA languages is addressed by a generic representation of model data while the data-flow analysis method enables an intuitive specification of analyses, which depends on the iterative propagation of results. We argued, that a traditional reachability analysis, which returns all direct and indirect neighbors of an element, is not suitable in the EA context and therefore has to be extended with context-sensitive propagation rules. For this purpose, we defined a relationship classification according to their semantics with respect to the architecture by defining the classes *located at*, *provides*, *consumes*, *structurally dependent* and *behaviorally dependent* (Sect. 4.1).

For each class, we defined rules for the propagation of change and failure impacts as well as for the quantification of the failure impact. Effect propagation depends on both the impact type and the semantics of the relationships connecting the respective elements. While this kind of analysis is easy to adopt in an existing EA ecosystem, the results may be imprecise. For a more focused analysis, we propose an analysis that relies on the classification of relationship types which are based on combinations of the impact types (for example *strong*, *weak* and *no effect*). This leads to slightly more complex adaptations but also to a more precise result. For both cases, we defined propagation rules in the form of DFA equations for best and worst case analysis. By extending these definitions, it is possible to include support for organization-specific semantics and additional relationship types. Executing the analysis using the DFA solver of the MAF framework yields the results which can be interpreted as estimations that reflect the best and worst case of the actual impact.

The combination of the generic model representations, extensible DFA-based analysis specifications and the classification approach for relationships ensures that this technique can be applied to the various EA conventions found in different organizations, even at the beginning of an EA initiative where little data is available. Further work has to be done to determine a suitable visualization of the results. Additionally, a broader evaluation of the relationship classifications has to be carried out, to verify their usability in practice. Field studies about the impact of changes in EA are required to determine suitable values for change probabilities. Future work should also develop a more abstract specification language to enable Enterprise Architect to customize the rule sets.

Acknowledgements. This work was partially sponsored by the FuE-Programm Informations- und Kommunikationstechnik Bayern. The authors would like to thank MID GmbH for providing their demo use case, licenses for their tool as well as for their support during the implementation.

References

1. Närman, P., Buschle, M., Ekstedt, M.: An enterprise architecture framework for multi-attribute information systems analysis. *Softw. Syst. Model.* **13**(3), 1085–1116 (2014)
2. Niemann, K.D.: *From Enterprise Architecture to IT Governance*. Springer, Heidelberg (2006)
3. Jonkers, H., Iacob, M.E.: Performance and cost analysis of service-oriented enterprise architectures. In: *Global Implications of Modern Enterprise Information Systems: Technologies and Applications*. IGI Global (2009)
4. Matthes, F., Monahov, I., Schneider, A., Schulz, C.: *EAM KPI Catalog v 1.0*. Technical report, Technical University Munich (2012)
5. de Boer, F., Bonsangue, M., Groenewegen, L., Stam, A., Stevens, S., van der Torre, L.: Change impact analysis of enterprise architectures. In: *IEEE International Conference on Information Reuse and Integration, Conference, IRI 2005*, pp. 177–181, August 2005
6. Kurpjuweit, S., Aier, S.: Ein allgemeiner Ansatz zur Ableitung von Abhängigkeitsanalysen auf Unternehmensarchitekturmodellen. In: *Wirtschaftsinformatik Proceedings 2009*, January 2009
7. Böhner, S.: Software change impacts - an evolving perspective. In: *Proceedings of International Conference on Software Maintenance*, pp. 263–272 (2002)
8. Lehnert, S.: A review of software change impact analysis. Ilmenau University of Technology. Technical report (2011)
9. Briand, L., Labiche, Y., O’Sullivan, L.: Impact analysis and change management of UML models. In: *Proceedings of International Conference on Software Maintenance ICSM*, pp. 256–265 (2003)
10. von Kneten, A., Grund, M.: QuaTrace: a tool environment for (semi-) automatic impact analysis based on traces. In: *Proceedings of International Conference on Software Maintenance ICSM*, pp. 246–255 (2003)
11. Saat, J.: Zeitbezogene Abhängigkeitsanalysen der Unternehmensarchitektur. *Multikonferenz Wirtschaftsinformatik* **2010**, 29 (2010)
12. Holschke, O., Närman, P., Flores, W.R., Eriksson, E., Schönherr, M.: Using enterprise architecture models and bayesian belief networks for failure impact analysis. In: *Feuerlicht, G., Lamersdorf, W. (eds.) ICSOC 2008*. LNCS, vol. 5472, pp. 339–350. Springer, Heidelberg (2009)
13. Tang, A., Nicholson, A., Jin, Y., Han, J.: Using bayesian belief networks for change impact analysis in architecture design. *J. Syst. Softw.* **80**(1), 127–148 (2007)
14. Buckl, S., Ernst, A., Lankes, J., Matthes, F.: *Enterprise architecture management pattern catalog (version 1.0)*. Technical report TB 0801. Technical University Munich, Chair for Informatics 19 (2008)
15. Jagt, R.: Support for multiple cause diagnosis with bayesian networks. Master of science thesis, Delft University of Technology, the Netherlands and Information Sciences Department, University of Pittsburgh, PA, USA, Pittsburgh (2002)

16. Kumar, A., Raghavan, P., Ramanathan, J., Ramnath, R.: Enterprise interaction ontology for change impact analysis of complex systems. In: IEEE Asia-Pacific Services Computing Conference APSCC, pp. 303–309 (2008)
17. Aryani, A., Peake, I., Hamilton, M.: Domain-based change propagation analysis: an enterprise system case study. In: IEEE International Conference on Software Maintenance (ICSM), pp. 1–9 (2010)
18. Lankhorst, M.: Enterprise Architecture at Work. Springer, GmbH & Co. KG, Heidelberg, Berlin (2012)
19. Saad, C., Bauer, B.: The model analysis framework - an IDE for static model analysis. In: Proceedings of the Industry Track of Software Language Engineering (ITSLE) in the Context of the 4th International Conference on Software Language Engineering (SLE 2011), May 2011
20. Saad, C., Bauer, B.: Data-flow based model analysis and its applications. In: Moreira, A., Schätz, B., Gray, J., Vallecillo, A., Clarke, P. (eds.) MODELS 2013. LNCS, vol. 8107, pp. 707–723. Springer, Heidelberg (2013)
21. The Open Group: ArchiMate 2.0 specification: Open Group Standard. Van Haren Publishing (2012)
22. U.S. Department of Defense: The DoDAF Architecture Framework Version 2.02 (2010). <http://dodcio.defense.gov/dodaf20.aspx>. Accessed 15 March 2015
23. MID GmbH: MID Innovator for Enterprise Architects (2014). <http://www.mid.de/produkte/innovator-enterprise-modeling.html>. Accessed 15 April 2014
24. Kienberger, J., Minnerup, P., Kuntz, S., Bauer, B.: Analysis and validation of AUTOSAR models. In: Proceedings of 2nd International Conference on Model-Driven Engineering and Software Development (2014)

<http://www.springer.com/978-3-319-20051-4>

Business Modeling and Software Design
4th International Symposium, BMSD 2014, Luxembourg,
Luxembourg, June 24-26, 2014, Revised Selected
Papers
Shishkov, B. (Ed.)
2015, XIII, 139 p. 48 illus., Softcover
ISBN: 978-3-319-20051-4