Supporting change propagation in the evolution of enterprise architectures

Hoa Khanh Dam, Lam-Son Lê and Aditya Ghose
School of Computer Science and Software Engineering
University of Wollongong
New South Wales 2522, Australia
{hoa, lle, aditya}@uow.edu.au

***Abstract*—Enterprise Architecture (EA) models the whole vision of an organisation in various aspects regarding both business processes and information technology resources. As the organisation grows, the architecture governing its systems and processes must also evolve to meet with the demands of the business environment. In this context, a critical issue is change propagation: given a set of primary changes that have been made to the EA model, what additional secondary changes are needed to maintain consistency across multiple levels of the EA. This paper proposes an enterprise architectural description language, namely ChangeAwareHierarchicalEA, integrated with a framework to support change propagation within an EA model. The core part of our change propagation framework is a new method for generating interactive repair plans from Alloy consistency rules that constrain the EA model.**

***Keywords*-enterprise architecture; change propagation; software evolution; enterprise architecture evolution**

I. Introduction

Enterprise Architecture (EA) captures the whole vision of an enterprise in various aspects regarding both business and information technology (IT) resources [1]. EA is a discipline that analyzes the services offered by an enterprise and its partners to the customer, the services offered by the enterprise to its partners, and the enterprise's organization and IT infrastructure. The representation of the enterprise can include various aspects such as enterprise's strategy, business services, business processes and web services.

In recent years, the ever-changing business environment demands constant and rapid evolution of an organisation. Consequently, changes to the EA models of such an organisation is inevitable if the EAs are to remain useful and to reflect the current state of organisational structures and business processes. For example, initial changes in an EA model which are made to an enterprise's strategy and business goals may lead to secondary changes made to the organisational structure of the enterprise. Such changes may lead to further changes in the business processes and services and so on. The ripple effect that an initial change may cause in enterprises is termed *change propagation*. In a large modern enterprise consisting of many elements, resources, business processes and infrastructures and their complex relationships, it becomes costly and labour intensive to correctly propagate changes. Unfortunately, there has been very little work on maintenance and evolution of enterprise architectures [2] [3]. Hence, we are in need of techniques and tools that provide more effective automated support for change propagation within an EA model. Change propagation may not be fully automated, since there are decisions that involve tradeoffs where human expertise is required. However, it is possible to provide tool support in tracking dependencies, determining what parts of the enterprise architecture are affected by a given change, and, as in this paper, determining and making secondary changes.

We aim to deal with change propagation which spans different hierarchical levels in an EA model. In order to achieve this, we define an enterprise architectural modeling language, namely ChangeAwareHierarchicalEA, which describes enterprise architecture in a hierarchical manner. This language is built on top of SeamCAD [4]. It extends SeamCAD in several ways: (1) making the modeling vocabulary easier to understand (2) lifting the limitation on cardinalities (3) addressing the well-formedness of services in an EA model. In addition, ChangeAwareHierarchicalEA specifically aims to support changes in the maintenance and evolution of enterprise architectures in a number of ways. Firstly, it offers an integrated view of the entire enterprise which allow us to, for example, analyze the effect at a business level of a change that takes place at a technical level. Secondly, it is supported by an (semi-)automatic mechanism of change propagation based on inconsistency management by extending an existing change propagation framework [**?**] to provide a method for generating repair options from Alloy consistency rules.

In the next section, we define the ChangeAwareHierarchicalEA modeling language and its toolkit. Section III then describes the change propagation framework followed by an example in Section IV. Related work is briefly presented in Section V. We conclude by discussing future directions in Section VI.

II. ChangeAwareHierarchicalEA: A Hierarchy-Oriented Modeling Framework for EA

In this section, we present a modeling framework that is developed as an improvement to an existing work (Subsection II-A). Our modeling framework consists of a modeling language definition (Subsection II-B) and a toolkit (Subsection II-C).

A. Hierarchy-Oriented EA Modeling

Modeling EA requires representing multiple diagrams of an enterprise, which typically shows the multiples business entities, IT systems and the services they offer. This could be done by a team of stakeholders having different backgrounds. One way to do this is to structure the model into hierarchical levels each of which can be of interest of just some, not all, stakeholders. Due to the differences in their background, stakeholders - the modelers may not want to use a single modeling approach, even a widely-recognized one, to build the enterprise model, which can be shared by the whole team. Developing a modeling framework that can be applied uniformly throughout the entire enterprise model and that can be used by all stakeholders is challenging. Firstly, the framework should have a uniform approach to specifying the services offered by business entities, IT systems and software components and to describing their implementation across hierarchical levels. Secondly, the framework should allow the stakeholders to represent the service specification and the service implementation of multiple business entities and IT systems, even within the same hierarchical level. Thirdly, the services offered by those entities and systems should be represented at different levels of granularity. Fourthly, the modeling framework should maintain the well-formedness of the enterprise model and the consistency between different diagrams opened by different stakeholders of the team.

SeamCAD was developed as a hierarchy-oriented modeling language and a computer-aided modeling tool that address the aforementioned challenges [4]. This modeling language allows the modeler to structure an enterprise into hierarchical levels, in terms of both organization and services. The computer-aided modeling tool helps the modeler visually build her model across levels and brings all levels together to make a coherent, well-formed model [5]. Enterprise models can be visually built and represented in a notation that is based on the Unified Modeling Language using this tool. The modeling language is formally defined in Alloy - a lightweight declarative language based on first order logic and set theory [6].

However, the SeamCAD modeling framework has several limitations. Firstly, the cardinality of containment relationship between a business entity (or an IT system) and its main modeling artifacts (e.g. properties, services) in SeamCAD is (exactly) one making it too simple to do modeling in practice. Secondly, the vocabulary used in SeamCAD is not natural for its potential users most of whom are in the community of EA, SOA or SE. The reason is that SeamCAD modeling terms [7] are originated from the Reference Model of Open Distributed Processing (RM-ODP)[1] - a theoretical standardization that supports principles in distributes processing, only one of which is about enterprise architecture. Thirdly, SeamCAD does not establish the correspondence between the sequence constraints of services of an entity or a system and those of collaborations in which the entity or system participates. Fourthly, the SeamCAD tool enforces all well-formedness rules defined in the SeamCAD modeling language making it inflexible to build EA models, particularly in the case where well-formnedness needs to be compromised for dealing with large models. These limitations were confirmed by the user's feedback collected from a total of 20 participants [4]. Therefore, we develop ChangeAwareHierarchicalEA as an improvement of SeamCAD while at the same time adding techniques to support change propagation in the evolution of modeling EA. The result is a new modeling framework, namely ChangeAwareHierarchicalEA.

Table I
INFORMAL DEFINITION OF CHANGEAWAREHIERARCHICALEA BUILDING BLOCKS

Building block	Informal Definition
SystemEntity	Represents any business unit, IT component or software component of the enterprise.
Stateful Property	Externally-observable properties that characterize a business entity or a system seen as whole. Can also be a parameter of services.
Stateless Property	Representation of the occurrence of a service. Stateless property can be considered as a context in which stateful properties are defined.
Service	Externally-observable service performed by a given business entity or a system seen as whole.
Collaboration	Interaction between multiple business entities or systems. The responsibility of each entity or system that gets involved in a collaboration is represented by a service.

B. Language Definition

The ChangeAwareHierarchicalEA modeling language has four main building blocks. They are informally defined in Table I. A model element of these building blocks can be viewed either as a whole or as a composite. A business entity or a system viewed as a composite has sub entities or sub systems. They interact with one another through collaborations. A business entity or a system viewed as a whole exhibits properties and offers services. In other words, if a business entity or a system is treated like white-box, it is viewed as a composite. If it is treated like black-box, it is seen as a whole. A property, a service or a collaboration can also be viewed as a composite, revealing component properties, services or a collaborations, respectively.

Relations are unchanged from SeamCAD to ChangeAwareHierarchicalEA. Both have participation link (between a business entity or a system to a collaboration),

[1]RM-ODP Resource website `http://www.rm-odp.net/`

association (between two properties), transition link (between two collaborations or between two services), generalization (between two model elements of the same building block), goal binding (between a service or a property and a collaboration) and means binding (between a collaboration and a service).

1) Meta-model: Figure 1 is a UML diagram that expresses the building blocks of the ChangeAwareHierarchicalEA modeling language. Two improvements over SeamCAD can be seen from this diagram: cardinality of association ends going to SystemEntity to other builing blocks is changed from one (`1`) to many (`0..*`); building blocks have new names.

The ChangeAwareHierarchicalEA modeling language come with a list of well-formedness rules. Most of them are taken from the SeamCAD modeling language. Three additional rules are defined for ChangeAwareHierarchicalEA. Two of them address the binding from services and properties of a business entity or system to collaborations in which it gets involved: if a business entity or system participates in a collaboration, it must offers a service and have property that is connected to the collaboration via goal binding. The third of these rules copes with how the sequences of services matched the sequences of collaborations.

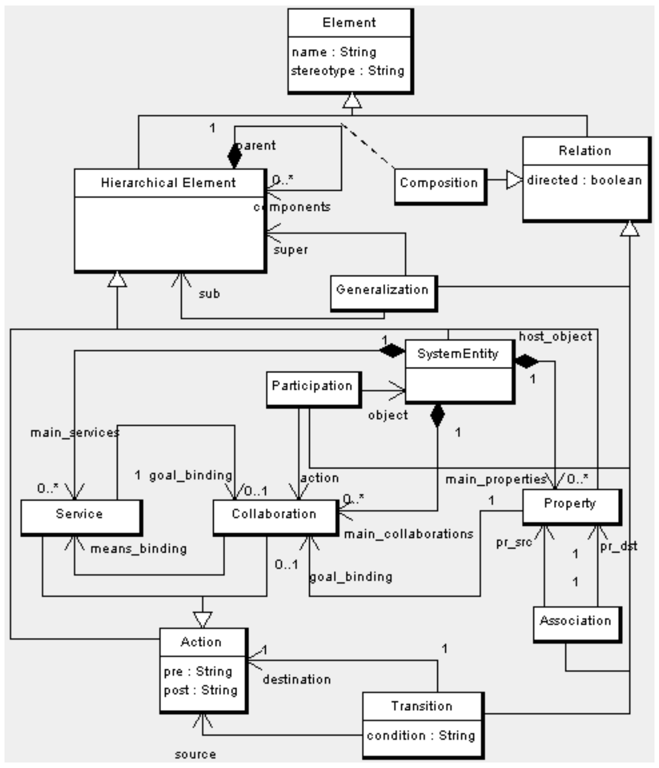

Figure 1. UML diagram expressing the ChangeAwareHierarchicalEA building blocks

2) Formalization in Alloy: The UML diagram shown in Figure 1 and the list of well-formedness rules can be formalized together in single Alloy code. As the Alloy language offers an object-oriented syntax, we can translate the UML diagram to Alloy straightforwardly as follows

- A UML class is mapped to an Alloy signature (the `sig` keyword)
- A UML role name is mapped to an Alloy field (to be declared within a signature)
- The UML cardinalities `1`, `0..1`, `1..*` and `0..*` are mapped to the `one`, `lone`, `some` and `set` keywords of Alloy, respectively
- The UML generalization is mapped to the extension mechanism in Alloy with the `extends` keyword
- For the sake of simplicity, we can ignore UML attributes of which types are primitives (e.g. `Date`, `String`) because they are not referred to in the well-formedness rules of ChangeAwareHierarchicalEA.

For example, the `SystemEntity` building block is formalized in the following Alloy signature.

```
sig SystemEntity extends HierarchicalElement {
   main_collaborations: set Collaboration,
   main_properties : set Property,
   main_services : set Service
}
```

Translating the well-formedness rules of ChangeAwareHierarchicalEA to Alloy code can be done in two steps: write first-order logic [8] statements for the BSDL description rules before mapping these statements to Alloy facts (the `fact` keyword). For instance, the rule that specifies goal bindings between a service offered by a business entity or a system and a collaboration it gets involved in can be translated to the following Alloy formula

```
fact responsibility {
all r: Participation | some s: Service |
            s.goal_binding = r.action
}
```

The Alloy code that formalizes the building blocks and well-formedness rules of ChangeAwareHierarchicalEA has a total of 122 lines of code that includes 11 Alloy signatures and 12 Alloy facts.

C. Toolkit

The ChangeAwareHierarchicalEA framework comes with a toolkit[2] that implements the ChangeAwareHierarchicalEA modeling language presented in previous subsections. We will describe an example (Subsection II-C1) before presenting how the tool works (Subsection II-C2). We also discuss how the tool manages the well-formedness of enterprise models and impact of changes potentially made by tool users (Subsection II-C3).

1) Example: Let us consider an example of a bookstore whose management decides to provide the companys services via the Internet. The management has a goal to specify the services that the bookstore can provide its customers with and to describe how to implement them

[2]The ChangeAwareHierarchicalEA toolkit is available at `http://www.uow.edu.au/~lle/ChangeAwareHierarchicalEA/index.htm`

using business and IT resources. A book-selling market contains a `BookValueNetwork` and a `Customer`. The value network consists of three companies: a bookstore company named `BookCo` (responsible for the service of processing the orders placed by the customer), a shipping company called `ShipCo` (responsible for shipping the books ordered) and a publishing company `PubCo` (responsible for supplying the books that were ordered but not yet available in the inventory of the bookstore company). The departmental structure of the bookstore company shows two departments: one for coping with the purchasing data (`PurchasingDep`) and the other for managing an inventory of books (`WarehouseDep`). We may have an addditional level showing IT infrastructure of these departments.

2) Diagrammatic Representation: Figure 2 is a screenshot of the ChangeAwareHierarchicalEA tool that illustrates how the Bookstore enterprise model is edited and visualized. As can be seen from this screenshot, each window of the tool features a tree view (to the top-left), a property view (to the bottom-left) and a diagram view (to the right). The tree-view displays the hierarchical structure of the whole enterprise model. In the diagram view, part of the bookstore enterprise model is a diagrammatically represented. In fact, the screenshot of Figure 2 shows the first hierarchical levels of the bookstore enterprise model. Note that `BookSellingMarket` is viewed as a composite whereas `BookValueNetwork` is seen as a whole. `sale` is a collaboration between `BookValueNetwork` and `Customer`. `BookValueNetwork` offers services `Sell`. `SellTxn` is a property that represents the execution context of the `Sell` service. `BookValueNetwork` has other properties that represent the money balance it has, the book it sells and information related to the booking-selling order it processes.

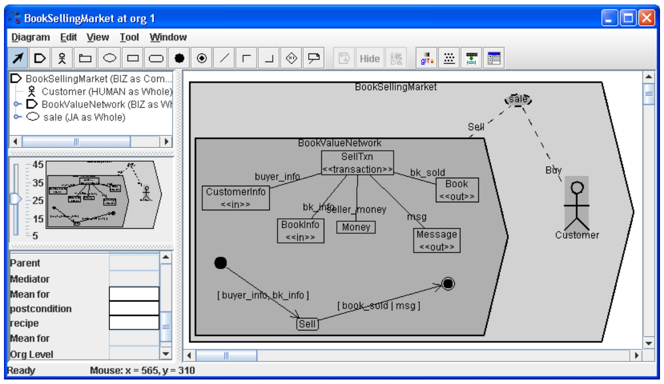

Figure 2. `BookValueNetwork`, `Customer` and their collaboration

The tool allows EA modelers to open multiple diagrams at the same time. Figure 3 is another diagram rendered in the tool. It captures the second hierarchical level of the bookstore enterprise model in which the three companies `BookCo`, `ShipCo` and `PubCo` are seen as whole. Another entity is represented at this hierarchical level is `Bank`

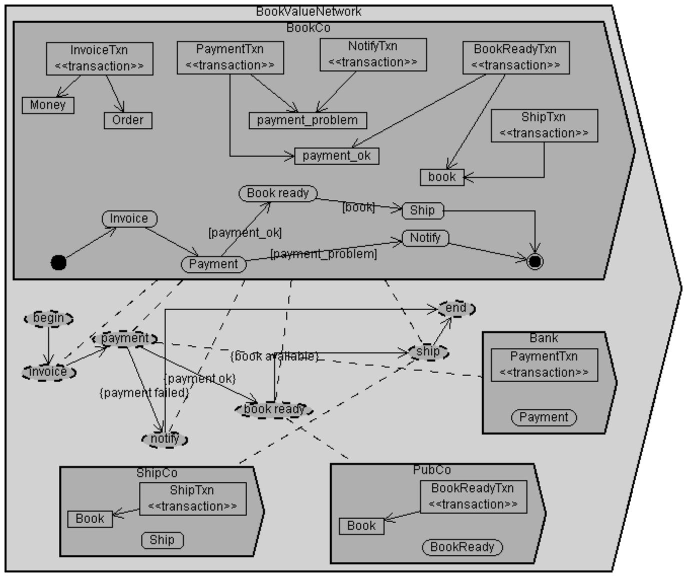

Figure 3. `BookCo`, `ShipCo`, `PubCo`, `Bank` and their collaboration. `Customer` is not in the scope of this diagram.

that processes payment orders (e.g. by clearing customer's cheques).

3) Well-formedness and impact of change: As an instance model of ChangeAwareHierarchicalEA, the bookstore enterprise model should conform to the well-formedness rules mandated in the ChangeAwareHierarchicalEA modeling language. For instance, `BookCo`, `ShipCo` and `PubCo` offer services according to how they get involved in `invoice`, `payment`, `book ready`, `notify` and `ship`. (In fact, this is constrained by the well-formedness rule presented in the first row of Table III.)

However, changes made to the bookstore enterprise model can potentially break this conformance. For example, adding a new or deleting an existing collaboration and a link going from `BookCo` to the collaboration may leave the corresponding service of `BookCo` unbound (i.e. the service has no goal binding). We need to make additional changes to keep the bookstore model well-formed. In other words, changes need to properly propagated across the bookstore model to preserve its well-formedness. In the next sections, we present and illustrate how changes are propagated in ChangeAwareHierarchicalEA models.

III. Change Propagation

The main purpose of change propagation process is to reintroduce consistency into the EA model by keeping track of the inconsistencies and making addition, secondary changes to repair these inconsistencies. Consistency is defined using a metamodel (of the EA model) and consistency rules, which define conditions that all models must satisfy for them to be considered valid. Those conditions may include, for example, syntactic well-formedness, coherence between different EA diagrams, and even best practices.

ChangeAwareHierarchicalEA provides support for change propagation using a generic framework which has been previously developed by the first author [?]. This framework provides a "change propagation assistant" that helps a modeler by suggesting additional (secondary) changes once primary changes have been made. Previous work has shown the effectiveness of this framework in supporting change propagation within agent-oriented design models [9–11]. Figure 4 shows an overview of how this change propagation framework is adopted in ChangeAwareHierarchicalEA.

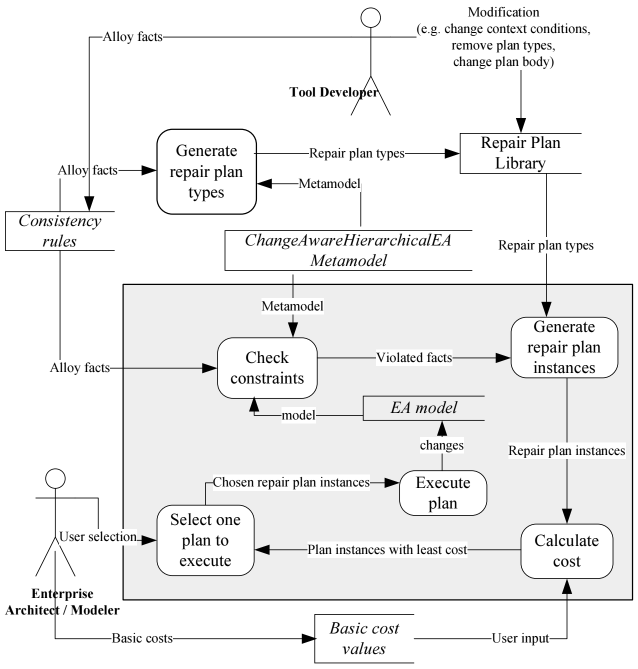

Figure 4. Change propagation framework for ChangeAwareHierarchicalEA

The key data items we deal with are a collection of well-formedness *rules*, an EA *model*, and a collection of *repair plans*. In ChangeAwareHierarchicalEA, rules are expressed in terms of Alloy facts. Repair plans represent different ways of fixing such rules's violations and are automatically generated ahead of time. Such repair plans form a library of repair plan types which will be instantiated at run time. Previous work [11] focuses on repair plan generation for rules expressed using the Object Constraint Language (OCL). In this work, we propose the automatic generation of repair plans for rules expressed as Alloy facts.

As soon as the enterprise architects make some primary changes to their enterprise model, they want to trigger the change propagation process (described as the shaded area in figure 4). Their model is then checked for inconsistencies by evaluating the consistency rules, which may result in the detection of a number of violated rules. When such a violation occurs, in order to have the rule violation repaired a corresponding event is generated. A given violation repairing event may trigger a number of possible repair plans instances (instantiated from the library of repair plans). The determination as to which repair plans to choose is generally a design decision, which can be dependent on various factors such as the cause of inconsistencies, or even factors other than consistency that contribute to a good design (e.g. experience, knowledge on the future evolution of the design, design styles). In general, many of these dependencies may not even be capable of being formulated formally and being captured without extra knowledge provided by the modeler. As a result, it is expected that the execution of repair actions requires user interaction.

In some cases, the number of different ways of fixing a inconsistency can be very large. Therefore, it is also important not to overwhelm the user with a large number of choices. For example, it is necessary to prevent infeasible repair options (e.g. repair actions that result in infinite cycles) from being presented to the user. The issue of repair plan selection has been addressed in [9] by defining a suitable notion of *repair plan cost* that takes into account the important cascading nature of change propagation and fixing inconsistencies. More specifically, the change propagation framework has a cost calculation component that is responsible for calculating the cost of each repair plan instance. It takes into account the fact that fixing one violated rule may also repair or violate others as a side effect, and so the cost calculation algorithm computes the cost of a given repair plan instance as including the cost of its actions (using basic costs defined by the modeler), the cost of any other plans that it invokes directly, and also the cost of fixing the violation of any rules that are made false by executing the repair plan. The user may use this mechanism to adjust the change propagation process. For example, if he/she wishes to bias the change propagation process towards adding more information then he/she may assign lower costs to actions that create new entities or add entities, and higher costs to actions that delete entities. The least cost plans (which can be more than one) are presented to the modeler for selection. It is also noted that the user can choose not to make any selection, in this case he/she continues performing further primary changes and invokes the change propagation process later.

A. Generate repair plans

Options for repairing inconsistencies ("repair plans") are represented using event-triggered plans. The syntax for repair plans is formally defined in figure 5. Each repair plan, P, is of the form $E : C \leftarrow B$ where E is the triggering event; C is an optional "context condition" (Boolean formula); and B is the plan body [11]. Triggering events can be one of the following four types: making a fact F true (F_{true}), or making a fact F false (F_{false}), or adding an entity to a derived set, or removing it from a derived set. Addition and deletion for derived sets (e.g. union sets) are represented as events because they do not usually involve a primitive action. Rather, they require a number of different actions, for example two deletions might be needed, one of the first set and the other for the second set, in the case of removing an entity from a union of two sets. A repair plan's context condition specifies when the plan should be applicable[3] and can be expressed in terms of first-order logic formulaes[4]). The context condition also contains a predicate $Ask(userVal, guidance)$ indicating that the user should be asked to provide (following the hints provided in the *guidance*) a value bounded to *userVal*. For example, the user is asked to provide a value for a new attribute. It is also noted that additional criteria can be incorporated in the context condition such as design heuristics, user preferences, etc.

The plan body can contain primitive repair actions, sequences ($B_1;\ B_2$), events which will trigger further plans (written as $!E$), conditionals and loops. There are several types of primitive repair actions. "Create $e : t$" indicates a creation of entity e of type t whilst "Delete e" results in a deletion of entity e. "Connect $e1$ and $e2$ (w.r.t. r)" denotes a connection between entities $e1$ and $e2$ with respect to relationship r, and similarly "Disconnect $e1$ and $e2$ (w.r.t. r)" refers to a disconnection between entities $e1$ and $e2$ with respect to relationship r. "Change *attr* of *e* to *val*" involves modifying attribute *attr* of entity *e* to a new value *val*.

$$
\begin{array}{lll}
P & ::= & E[:\ C] \leftarrow B \\
E & ::= & F_{true} \mid F_{false} \mid \mathit{Add\ x\ to\ SE} \mid \mathit{Remove\ x\ from\ SE} \\
C & ::= & C \vee C \mid C \wedge C \mid \neg\ C \mid \\
 & & \forall x \bullet C \mid \exists x \bullet C \mid Prop \mid Ask(userVal,\ guidance) \\
B & ::= & RepairAction \mid\ !E \mid B_1;\ B_2 \mid \\
 & & \mathit{if\ C\ then\ B} \mid \mathit{for\ each\ x\ in\ SE\ B} \\
RepairAction & ::= & \mathit{Create}\ \mathrm{e : t} \mid \mathit{Delete}\ \mathrm{e} \mid \\
 & & \mathit{Connect}\ \mathrm{e1}\ \mathit{and}\ \mathrm{e2}\ (\mathit{w.r.t}\ \mathrm{r}) \mid \\
 & & \mathit{Disconnect}\ \mathrm{e1}\ \mathit{and}\ \mathrm{e2}\ (\mathit{w.r.t}\ \mathrm{r}) \mid \\
 & & \mathit{Change}\ \mathrm{attr}\ \mathit{of}\ \mathrm{e}\ \mathit{to}\ \mathrm{val}
\end{array}
$$

Figure 5. Repair plan abstract syntax

As a step towards automated change propagation, the repair plans in our approach are generated automatically (at design time) from the Alloy facts, and form a *repair plan library* which is used at run time. A key consequence of generating plans from facts, rather than writing them manually, is that, by careful definition of the plan generation scheme, it is possible to guarantee that the plans generated are correct, complete, and minimal, i.e. there are no repair plans to fix a violation of a fact other than those produced by the generator; and any of the repair plans produced by the generator can fix a violation. However, we also allow the users to use their domain knowledge and expertise to modify generated repair plans or remove plans that should not be executed.

Since ChangeAwareHierarchicalEA uses Alloy to specify its well-formedness rules, we have developed a translation that takes an Alloy fact as input and generates repair plans that can be used to correct fact violations. Such a translation can be developed by considering all the possible ways in which a rule can be false, and hence all the possible ways in which it can be made true. However, there may be a large number of concrete ways of fixing a rule violation. In some cases this number can be infinite. For instance, consider a fact requiring a particular set *SE* to be non-empty. Assume that *SE* is empty, then the various ways of fixing this fact are adding 1 element, adding 2 elements, adding 3 elements, and so on. Another example is a fact requiring that the age (attribute) of a person (model entity) has to be greater than 18. There is also an infinite number of ways to fix this rule, each of which corresponds to changing the age to a number greater than 18. Such issues are due to the inherent characteristics of first order logic that Alloy facts is based on. In order to address those issues, our repair plan generation has the following important characteristics. Firstly, generated repair plans abstractly represent certain classes of concrete ways of fixing a rule. For example, plan $c_{true} : x \in SE \leftarrow !c1_{true}(x)$ represents all the repair plans that make rule c true, each of which corresponds to picking an element x in the set SE and making $c1(x)$ true. Secondly, generated repair plans are minimal in that all of their actions contribute towards fixing a certain rule, i.e. taking out any of the actions results in failing to fix the rule. For example, the generated repair plan for making a (empty) set be not empty is adding only one element to the set. Finally, there is a type of repair plan that involves user interaction in which the user is asked to provide an input. For example, in the above example the user is asked to input an age value that is greater than 18. A proof for showing the correctness and completeness of generated repair plans for rules expressed in OCL is presented in [11]. Our future work would involve developing a similar proof for Alloy rules.

Our repair plan generation for ChangeAwareHierarchicalEA contain rules which cover a wide range of Alloy expressions. Due to space limitation we present here an excerpt of the rules in Table II.

IV. Illustration of Change Propagation

In order to illustrate how ChangeAwareHierarchicalEA deals with changes, we are developing a case study, an excerpt of which is presented in this section. The full case study is a Bookstore which is specified and designed, along

[3]In fact when there are multiple solutions to the context condition, each solution generates a new plan instance. For example, if the context condition is $x \in \{1, 2\}$ then there will be two plan instances.

[4]"Prop" denotes a primitive condition such as checking whether $x > y$ or whether $x \in SE$.

Table II
PLAN GENERATION RULES FOR BASIC ALLOY FACTS

Alloy facts	**Repair plans**
`fact F{e1.aend = e2}`	F_{true} : e1.aend = null $\leftarrow$ Connect e1 and e2 (w.r.t *aend*) F_{true} : e1.aend = x $\leftarrow$ Disconnect e1 and x (w.r.t *aend*); Connect e1 and e2 (w.r.t. *aend*)
`fact F{e1.aend1 = e2.aend2}`	F_{true} : e2.aend2 = x $\leftarrow$ $!(e1.aend1 = x)_{true}$ F_{true} : e1.aend1 = x $\leftarrow$ $!(e2.aend2 = x)_{true}$
`fact F{all x : SE | F1}`	F_{true} $\leftarrow$ for each x in SE if $\neg$ F1(x) then $!F'_{true}$(x) F'_{true}(x) $\leftarrow$ $!(RemovexfromSE)$ F'_{true}(x) $\leftarrow$ $!F1_{true}$(x)
`fact F{some x : SE | F1}`	F_{true} : x $\in$ SE $\leftarrow$ $!F1_{true}$(x) F_{true} : x $\in$ Type(SE) $\wedge$ x $\notin$ SE $\leftarrow$!(Add x to SE) ; $!F1_{true}$(x) F_{true} $\leftarrow$ Create x : Type(SE) ; !(Add x to SE) ; $!F1_{true}$(x)
`fact F{F1 and F2}`	F_{true} : $\neg$ F1 $\wedge$ F2 $\leftarrow$ $!F1_{true}$ F_{true} : $\neg$ F2 $\wedge$ F1 $\leftarrow$ $!F2_{true}$ F_{true} : $\neg$ F1 $\wedge$ $\neg$ F2 $\leftarrow$ $!F1_{true}$; $!F2_{true}$
`fact F{F1 or F2}`	F_{true} $\leftarrow$ $!F1_{true}$ F_{true} $\leftarrow$ $!F2_{true}$

with a number of additional requirements. These additional requirements give examples of evolution of the enterprise architecture that are used to assess how ChangeAwareHierarchicalEA supports change propagation.

The initial architecture of the Bookstore has been presented earlier in Subsection II-C1. We assume that the bookstore's management wants to change the payment methods supported by the bookstore from taking customer's cheques to accepting customer's credit card. This change at high-level business requirements leads to changes made to the bookstore enterprise model. Now, the three companies `BookCo`, `ShipCo`, `PubCo` need to collaborate with an agent (called `CreditCard Agent`) who is capable of verifying credit cards and processing credit transactions. In addition, the business of verifying credit cards and processing credit transactions are now represented as component collaborations of the `payment` collaboration. Note that the `notify` collaboration now becomes a component collaboration of `payment`. Figure 6 is a diagram that shows the second hierarchical level of the bookstore enterprise model after these initial changes are made. In this diagram, added elements are marked with tick symbols.

After making such initial changes, the enterprise architect may wish to ask the tool what other entities they should alter to correctly propagate the new change. In fact, the removal of `Bank`, the addition of `CreditCard Agent`, `verify cc`, `transfer money` and the relocation of `notify` break the well-formedness of the bookstore enterprise model. Specifically, these changes violate a total of three well-formedness rules defined in the ChangeAwareHierarchicalEA modeling language. Table III explains these rules both informally (in English sentences) and formally (by means of Alloy code).

The underlying change propagation process in ChangeAwareHierarchicalEA firstly checks for rule violations in the new EA model. As mentioned earlier, the first rule is violated in several instances. As can be seen from figure 6, the new business entity `CreditCard`

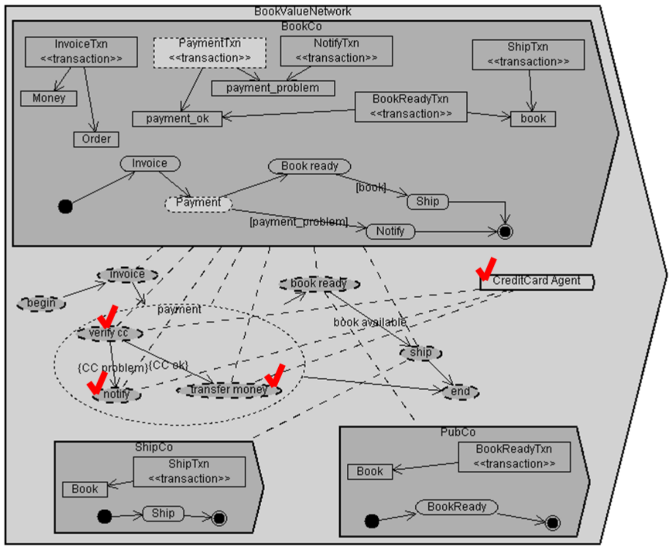

Figure 6. Representation of the bookstore enterprise model at the same scope as Figure 3 after initial changes are made.

`Agent` gets involved in the new collaboration `verify cc` (similarly to the two other new collaborations `transfer money`, `notify`) but it does not offer any service that is bound to the collaboration. Therefore, the first rule is violated for the participation between `CreditCard Agent` and `verify cc`, denoting `CreditCardAgent_verifycc`.

To fix this rule violation, ChangeAwareHierarchicalEA uses and instantiates repair plans which have been generated earlier. For instance, let us have a look at the first consistency rule which is expressed as an Alloy fact as follow.

fact F $\{$*all r : Participation* | *some s : Service* | *s.goal_binding = r.action*$\}$

We define the following abbreviations:

fact F1(r) $\{$*some s : Service* | *s.goal_binding = r.action*$\}$

fact F2(r, s) $\{$*s.goal_binding = r.action*$\}$

ChangeAwareHierarchicalEA produces the following re-

Table III
WELL-FORMEDNESS RULES THAT ARE VIOLATED BY INITIAL CHANGES MADE TO THE BOOKSTORE ENTERPRISE MODEL

Rule	**Formalization in Alloy**

A business entity or a system that gets involved in a collaboration must offer a service that is bound to the collaboration.	`fact responsibility {`
	`all r: Participation |`
	`some s: Service |`
	`s.goal_binding = r.action }`

The orders in which services are performed within a business entity comply with the orders that is specified for collaborations the business entity gets involved in.	`fact sequence {`
	`all s1, s2: Service |`
	`s2 in s1.^(~source.destination)`
	`=> (some c1, c2: Collaboration |`
	`s1.goal_binding = c1 and`
	`s2.goal_binding = c2 and`
	`c2 in c1.^(~source.destination)) }`

The decomposition hierarchy of services must be the same as the decomposition hierarchy of collaborations that they are bound to.	`fact binding_hierarchy {`
	`all c1, c2: Collaboration |`
	`all s1, s2: Service |`
	`(c2 in c1.components and`
	`s1.goal_binding = c1 and`
	`s2.goal_binding = c2)`
	`=> s2 in s1.components }`

pair plans for making fact F true, since it has the form $all\, x : SE \mid F1$

$F_{true} \leftarrow$ for each r in Participation
  if $\neg$ F1(r) then $!F'_{true}$(r) **(P1)**
F'_{true}(r) $\leftarrow$ Remove r from Participation **(P2)**
F'_{true}(r) $\leftarrow$ $!F1_{true}$(r) **(P3)**

For the $F_1(r)$ fact, the following repair plans are generated, since the rule is of the form *some* $x : SE \mid F1$. In the rules of Table II, "Type(SE)" denotes the type of SE's elements. In this case, SE (which is Service) contains all the available services and therefore the first and second plan are the same and creating a new service also adding it to the set of services.

$F1_{true}(r)$: s $\in$ Service $\leftarrow$ $!F2_{true}$(r, s) **(P4)**
$F1_{true}(r)$ $\leftarrow$ Create s : Service ; $!F2_{true}$(r, s) **(P5)**

Similarly for fact $F_2(r,s)$, the following repair plans are generated.

$F2_{true}(r,s)$: r.action = x $\leftarrow$ $!(s.goal\_binding = x)_{true}$ **(P6)**
$F2_{true}(r,s)$: s.goal_binding = x $\leftarrow$ $!(r.action = x)_{true}$ **(P7)**

Finally, the following repair plans are generated to make $s.goal\_binding = x$ and $r.action = x$ true.

$(s.goal\_binding = x)_{true}$: s.goal_binding = null
  $\leftarrow$ Connect s and x (w.r.t *goal_binding*) **(P8)**
$(s.goal\_binding = x)_{true}$: s.goal_binding = y
  $\leftarrow$ Disconnect s and y (w.r.t *goal_binding*);
    Connect s and x (w.r.t. *goal_binding*) **(P9)**
$(r.action = x)_{true}$: r.action = null
  $\leftarrow$ Connect s and x (w.r.t *action*) **(P10)**
$(r.action = x)_{true}$: r.action = y
  $\leftarrow$ Disconnect s and y (w.r.t *action*);
    Connect s and x (w.r.t. *action*) **(P11)**

As discussed earlier in Section III, ChangeAwareHierarchicalEA allows users to use their domain knowledge and expertise to modify generated repair plans or remove plans that should not be executed. For instance, plan P2 can be removed since we do not want to remove a Participation. In addition, in order to make sure that the service and participation are associated to the same business entity, the user may change the context condition of plan P4 and the body of plan P5 as follows.

$F1_{true}(r)$: s $\in$ Service $\wedge$ s.host_object = r.object
  $\leftarrow$ $!F2_{true}$(r, s) **(P4$_{new}$)**
$F1_{true}(r)$ $\leftarrow$ Create s : Service ;
  Add s to r.object; $!F2_{true}$(r, s) **(P5$_{new}$)**

Figure 7 represents the event-plan tree for fixing the violation of fact F using the plans produced by our repair plan generator. At the root of the tree is the event of making the top constraint fact F true. The leaves of the tree are primitive repair actions.

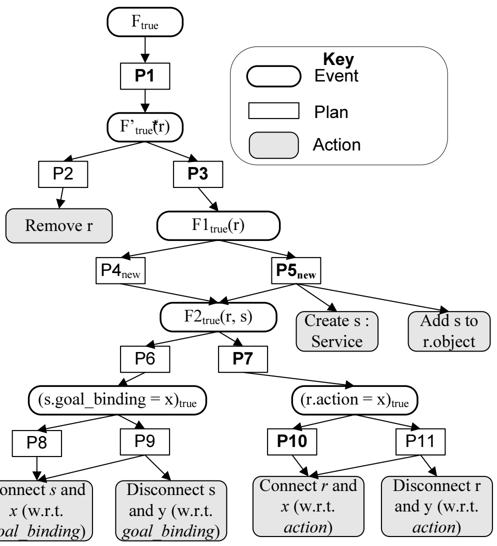
F_true
P1
F'_true(r)
P2
P3
Remove r
F1_true(r)
P4_new
P5_new
F2_true(r, s)
Create s : Service
Add s to r.object
P6
P7
(s.goal_binding = x)_true
(r.action = x)_true
P8
P9
P10
P11
Connect s and x (w.r.t. goal_binding)
Disconnect s and y (w.r.t. goal_binding)
Connect r and x (w.r.t. action)
Disconnect r and y (w.r.t. action)
Key
Event
Plan
Action

Figure 7. An event-plan tree for F$_{true}$

Repair plan P4 is not applicable since `CreditCard Agent` does not have any services at the time. Hence, the only applicable repair plan for making F1 true is P5, which create a new service in `CreditCard Agent` and making F2 true for the newly-created service. Two repair plan types P6 and P7 which can potentially make F2 true for the new service. However, plan P7 is not applicable in this case since there is no collaboration to which the newly-created service is bound to (i.e. `s.goal_binding = null`). On the other hand, plan P6 is applicable since the associated collaboration of r is `verify cc`, i.e. `r.action = verify` cc. P6 potentially can trigger either plan P8 or P9. However, since the new service is currently not associated with any collaboration, i.e. s.goal_binding = null, only plan P8 is applicable in this case. Therefore, ChangeAwareHierarchicalEA will propose the following actions to fix the first rule:

1) Create a new CC Verification service
2) Add this new service into CreditCard Agent
3) Set verify cc to be the collaboration of the new service.

However, these changes will violate the other two rules listed in Table III, which lead to further changes being needed. In a similar manner, ChangeAwareHierarchicalEA uses and instantiates repair plans to fix those rule violation. As discussed in Section III, we calculate the cost for each repair plan and present a list of equal least cost plans to the user. The repair plan's cost is computed by firstly accumulating the costs of each repair actions in the plan. For instance, the basic cost of plan P1 for fixing the first rule violation would be the sum of the costs of a creation of a new entity, a connection between two entities and a modification of an entity's attribute. In addition, the cost of plan P1 includes the cost of fixing the other two rules' violation which it breaks. We follow the same process of generating plan instances to calculate the cost of fixing the violation of those rules. Due to space limitation, we do not include a full discussion here but provide a list of additional changes proposed by ChangeAwareHierarchicalEA, which are marked with star symbols (*), in Table IV.

Table IV
INITIAL CHANGES LEAD TO FURTHER CHANGES MADE TO THE BOOKSTORE ENTERPRISE MODEL

Category	Model element
Removal	Bank business entity
Addition	CreditCard Agent business entity
Addition	verify cc collaboration
Addition	transfer money collaboration
Mod	payment becomes parent of notify collaboration
Removal	Link Bank to payment
Addition	Link CreditCard Agent to verify cc
Addition	Link CreditCard Agent to transfer money
Addition	Link CreditCard Agent to notify
Removal	Partcipation link Bank to notify
Addition*	CC Verification service in BookCo
Addition*	CC Verification service in CreditCard Agent
Addition*	TransferMoney service in BookCo
Addition*	TransferMoney service in CreditCard Agent
Addition*	Notify service in CreditCard Agent
Addition*	Transition verify cc to notify
Addition*	Transition verify cc to transfer money
Removal*	Transition payment to notify
Addition*	Transition CC Verification to Notify
Addition*	Transition CC Verification to TransferMoney
Removal*	Transition Payment to Notify
Addition*	Payment service in CreditCard Agent
Mod*	Payment service becomes parent of CC Verification service in BookCo
Mod*	Payment service becomes parent of Transfer Money service in BookCo
Mod*	Payment service becomes parent of Notify service in BookCo
Mod*	Payment service becomes parent of CC Verification service in CreditCard Agent
Mod*	Payment service becomes parent of Transfer Money service in CreditCard Agent

V. Related Work

A number of methods have been developed for modeling enterprises. Archimate proposes an integrated modeling framework for EA including organizational structure, business processes, information systems and infrastructure [12]. The Computer Integrated Manufacturing Open System Architecture (CIMOSA, also known as the ISO EN/IS 19440 standard)[5] focuses on the modeling of processes in the context of computer integrated manufacturing projects. Design and Engineering Methodology for Organizations is a method for (re-)designing organizations [13]. IDEF[6] (Integrated DEFinition Methods) is a set of methods that address many aspects of enterprise modeling (e.g. function, data, process, object-oriented design). TOGAF[7] and Zachman [14] propose ad-hoc modeling frameworks in which multiple systems can be represented. The well-formedness of enterprise models is not specficially defined in these methods. It is up to the EA modelers to preserve the well-formedness of their models if they wish to.

There has been very little work to deal with change propagation in enterprise architecture. To the best of our knowledge, only Archimate was taken for analyzing change impact in EA [2]. This work helps EA modelers identify potential impacts of a change made to their enterprise model before it actually takes place. Their work proposes a classification of relationships between different entities within an EA model. Such relationships are however defined at a high level and it is not clear how they can be used to propagate changes. Another work in this area illustrates how to propagate changes and determine change impacts through rules and ontology [3]. This work does not address any particular EA framework. In addition, it does not deal with the issue of how to repair such rules when they are violated.

Several approaches provide developers with a software development environment which allows for recording, presenting, monitoring, and interacting with inconsistencies to help the developers resolve those inconsistencies [15]. Other work also aims to automate inconsistency resolution by having pre-defined resolution rules (e.g. [16]) or identifying specific change propagation rules for all types of changes (e.g. [17]). However, these approaches suffer from the correctness and completeness issue since the rules

[5]CIMOSA homepage http://www.cimosa.de/
[6]Integrated Definition Methods http://www.idef.com/
[7]The Open Group Architecture Framework http://www.togaf.org/

are developed manually by the user. As a result, there is no guarantee that these rules are complete (i.e. that there are no inconsistency resolutions other than those defined by the rules) and correct (i.e. any of the resolutions can actually fix a corresponding inconsistency). In order to deal with this issue, [18] has proposed an approach for automatically generating repair options by analyzing consistency rules expressed in first order logic and models expressed in xLinkIt. They did not take into account dependencies among inconsistencies and potential interactions between repair actions for fixing them. In other words, their work considers repair actions as independent events, and thus does not explicitly deal with the cascading nature of change propagation.

VI. Conclusions and future work

In this paper, we have presented a new enterprise architectural description language and toolkit, namely ChangeAwareHierarchicalEA, which can be used to describe enterprise architecture in a hierarchical manner. This language is built on top of SeamCAD and has several key improvements in terms of the understandability of the vocabulary, cardinalities, and well-formedness of services in an EA model. An important novel aspect of ChangeAwareHierarchicalEA resides in its capability to deal with changes made to an existing EA model. It is integrated with a framework which provides (semi-)automated support for propagating further changes across an EA model after certain initial changes have been made. The underlying change propagation framework of ChangeAwareHierarchicalEA automatically generates repair plans from consistency and well-formedness rules defined in Alloy. Such repair plans are the driver of propagating futher changes.

We have implemented ChangeAwareHierarchicalEA and started integrating the change propagation framework into its toolkit. Our future work also involves implementing a repair plan generator which covers a complete set (or a majority of) Alloy expressions. This will lead to the ability to deal with changes in the landscape of strategic alignment. The feasibility of this direction can be judged in the way that we use Alloy to formalize the meta-model of a specific modeling method and an accompanied toolkit that are particularly designed to address the strategic alignment of business services. Our future work also includes an evaluation to test the effectiveness and efficiency of ChangeAwareHierarchicalEA using real life models.

References

[1] J. Schekkerman, *How to Survive in the Jungle of Enterprise Architecture Framework: Creating or Choosing an Enterprise Architecture Framework*. Trafford Publishing, 2004.

[2] A. W. Stam, "Change Impact Analysis of Enterprise Architectures," in *Proceedings of the 2005 IEEE Conference on Information Reuse and Integration*. IEEE Computer Society, 2005, pp. 177–181.

[3] A. Kumar, P. Raghavan, J. Ramanathan, and R. Ramnath, "Enterprise Interaction Ontology for Change Impact Analysis of Complex Systems," in *Proceedings of the 2008 IEEE Asia-Pacific Services Computing Conference*. IEEE Computer Society, 2008, pp. 303–309.

[4] L.-S. Lê, "SeamCAD: a Hierarchy-Oriented Modeling Language and a Computer-Aided Tool for Enterprise Architecture," Ph.D. dissertation, EPFL, 11 2008.

[5] L.-S. Lê and A. Wegmann, "SeamCAD: Object-Oriented Modeling Tool for Hierarchical Systems in Enterprise Architecture," in *39th Hawaii International Conference on System Sciences*, vol. 8. IEEE Computer Society, 2006, p. 179c.

[6] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Trans Softw Eng Methodol*, vol. 11, no. 2, p. 256290, 2002.

[7] A. Wegmann, L.-S. Lê, G. Regev, and B. Wood, "Enterprise modeling using the foundation concepts of the RM-ODP ISO/ITU standard," *Information Systems and E-Business Management*, vol. 5, no. 4, pp. 397–413, 2007.

[8] R. Smullyan, *First-Order Logic*. Dover Publications, 1995.

[9] K. H. Dam. *Supporting Software Evolution in Agent Systems*. PhD thesis, RMIT University, School of Computer Science and IT, 2009.

[10] K. H. Dam and M. Winikoff, "Cost-based BDI plan selection for change propagation," in *Proceedings of the 7th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, Padgham, Parkes, Müller, and Parsons, Eds., Estoril, Portugal, May 2008, pp. 217–224.

[11] K. H. Dam, M. Winikoff, and L. Padgham, "An agent-oriented approach to change propagation in software evolution," in *Proceedings of the Australian Software Engineering Conference (ASWEC)*. IEEE Computer Society, 2006, pp. 309–318.

[12] K. H. Dam and M. Winikoff, "Generation of repair plans for change propagation," in *Agent-Oriented Software Engineering VIII*, ser. Lecture Notes in Computer Science, M. Luck and L. Padgham, Eds., vol. LNCS 4951. Springer Berlin / Heidelberg, April 2008, pp. 132–146.

[13] M. Lankhorst, *Modelling, Communication and Analysis*. Springer, 2005.

[14] J. Dietz, *Enterprise Ontology: Theory and Methodology*. Springer, 2006.

[15] J. A. Zachman, "A framework for information systems architecture," *IBM Syst. J.*, vol. 26, no. 3, pp. 276–292, 1987.

[16] J. Grundy, J. Hosking, and W. B. Mugridge, "Inconsistency management for multiple-view software development environments," *IEEE Transactions on Software Engineering*, vol. 24, no. 11, pp. 960–981, 1998.

[17] W. Liu, S. Easterbrook, and J. Mylopoulos, "Rule based detection of inconsistency in UML models," in *Proceedings of UML Workshop on Consistency Problems in UML-based Software Development*, 2002, pp. 106–123.

[18] L. C. Briand, Y. Labiche, L. O'Sullivan, and M. M. Sowka, "Automated impact analysis of UML models," *Journal of Systems and Software*, vol. 79, no. 3, pp. 339–352, March 2006.

[19] C. Nentwich, W. Emmerich, and A. Finkelstein, "Consistency management with repair actions," in *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*. IEEE Computer Society, 2003, pp. 455–464.