ON THE SEMANTICS OF ARCHITECTURAL DECISIONS

KRZYSZTOF SACHA

Warsaw University of Technology
Nowowiejska 15/19, Warszawa, 00-665, Poland
k.sacha@ia.pw.edu.pl

The architecture of a software system results from decisions made by the developers throughout the software life cycle. Any decision pertaining to software architecture is called an architectural decision. Architectural decision modelling captures the dependencies that exist between the decisions and serves as a foundation for knowledge management and reuse. Several models have been described in the literature, using natural language to explain the basic notions and class diagrams to show relations between them. However, a formal definition of an architectural decision is still missing. This paper analyzes existing architectural decision models and provides a formal background for the basic notions that all the models have consensus on. The major contribution of this paper is twofold: to propose a set-theoretic definition of the semantics of architectural decisions; and to show an explicit interpretation of basic relationships that exist in the architectural knowledge. The formalization can help in understanding the meaning of architectural decisions and the meaning of relations that exist between the decision elements. UML-based metamodel for architectural design decisions is also presented.

Keywords: Architectural decision; architectural knowledge; decision semantics; model; SOA.

1. Introduction

IEEE 1471 defines software architecture as the fundamental organization of a software system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution [1]; a similar definition is given in ISO/IEC/IEEE 42010 [2]. The architecture of software can be presented using UML diagrams, and described in an architectural description document, which captures and communicates the results of the design process. A broad scope architecture description can be organized into a set of views, each of which is a collection of models that represent one aspect of the entire system. A well-known example of such a partitioning of the architectural description is the 4+1 views model [3]. Other aspects of software, like the choreography of services in Service Oriented Architecture (SOA) can also be added.

Creating software architecture is not a single activity, but rather a long sequence of decisions made by the developers and other stakeholders throughout the software life cycle. Any decision pertaining to software architecture is called an architectural decision. No formal definition of this notion exists. The subject of architectural decisions is usually related to non-functional concerns such as performance, security or transactionality of services. However, functionality of software can also be taken into account, particularly in software evolution projects where the cost of changes to the existing systems may compromise the requirements.

Several models of architectural decisions have been described in the literature [4-12], illustrated by class diagrams to show basic notions and relations. Those models look similar but have some significant differences, as pointed out in Section 2. Because a formal – i.e., based on solid mathematical foundations – definition of an architectural decision is missing, several properties of the model elements cannot formally be decided. In this paper, we present a set-theoretic definition of the semantics of architectural decisions. The other contribution of this paper is a demonstration of the meaning of basic relationships that exist in the architectural knowledge. The formalization can help in deep understanding of the meaning of architectural decisions and of the relations that exist between the decision elements. The last part of the paper contains a meta-model based on the formal definitions, which bridges the gap between conceptual and practical aspects of the software architecture design process.

The remaining part of the paper is organized as follows. Section 2 discusses related work on modelling architectural decisions. The semantics of architectural decisions is defined in Section 3 and a formal architectural decision model is described in Section 4. Metamodel for architectural design decisions is outlined in Section 5. The conclusions and plans for further research are given in Section 6.

2. Related Work

Architectural decision modelling can be used to document decisions that have already been made. Text templates of attributes and illustrating diagrams are used to capture this knowledge [4,8,12-14]. More elaborate approaches focus on modelling architectural decisions and the relationships that exist between the decisions within the decision making process [5,6,10,11,15,16]. Tools to support particular approaches have been implemented [7-10,17-20]. A few surveys on methods and the existing tools are available in the literature [21-23].

An architectural decision model introduced in [6] classifies architectural decisions into several types, shows the lifecycle of these decisions, and defines the attributes that have to be captured before a decision can be made. The most important attributes are: *scope* (context) and *rationale* that stands behind the decision. An architectural decision is treated as an entity, which comprises both: A design problem and the solution. Alternative solutions to the design problem are modelled as alternative decisions, which can evolve along with the design process in a manner described by a state machine. The states can be, e.g., *idea*, *tentative*, *decided*, *approved* and *rejected*. In addition, a set of relationships between the decisions is defined, to express dependencies that exist in the architectural knowledge domain. The semantics of the model elements is defined informally using textual descriptions.

A distinction between an architectural problem (a decision to make with many options open) and its solution (a decision made) has been introduced in [5]. The separation of problems and solutions allows modelling the alternative solutions and the decisions made over time in a more clear way. However, the relationships that exist between different problems and different solutions are not defined in an explicit way. The semantics of architectural decisions and their relationships is defined informally using text and figures. Little effort is made towards reusability of architectural decision knowledge accumulated in the model.

A rich model described in [11] makes a clear distinction between an architectural problem (referred to as an issue), a set of possible alternative solutions (referred to as alternatives) and the solution selected (referred to as an outcome). Closely related issues can be grouped together into a hierarchy of topic groups. Topic groups, issues and the related alternatives are organized into levels of abstraction (conceptual, technology and vendor asset) and into layers corresponding to a SOA architecture model (process, service and integration). Relationships between alternatives, alternatives and issues, and issues alone are defined across the entire model. The resulting graph structure can show partial decision order. Metamodel describing concepts and their relations is defined as an UML class diagram [11,16]. Formal semantics of the model elements has not been provided. A significant effort towards reusability has been made, which resulted in a definition of Reusable Architectural Decision Model for SOA.

To summarize, the basic concepts of architectural design issue (a problem), an alternative (a possible solution), a rationale and the outcome (decision made on the problem) have consensus in all architectural decision models.

3. The Meaning of Architectural Decisions

The architecture of software is shaped by decisions, which are made to solve problems encountered in the software design process. For example, SOA systems can invoke services using synchronous or asynchronous programming styles. Which style to use is a problem, which has at least two solution options, each of which determines an architectural feature of the designed software system. Such an architectural option is called a *decision* with the status of *tentative* in [6], a *solution* in [5] and an *architectural decision alternative* in [11]. The problem itself is called a *problem* (solved by decisions) in [5] and an *architectural decision issue* (which contains architectural decision alternatives) in [11]. However, it is not defined explicitly in [6]; instead, a relation *IsAnAlternativeTo* between the decisions is used in order to group the options related to the same architectural feature.

To put things in order, we present in this section a set-theoretic semantics of architectural decisions. The formalization can help in deep understanding of the meaning of architectural decisions and of the relation between architectural decisions and the architecture of software.

Designing a software system is a decision process of finding an acceptable solution, which fulfils the requirements. Making a design decision is equivalent to excluding a group of systems, which do not comply with the decision, and narrowing the solution space down to those systems that match the decision. Having this in mind we can adopt the following set of basic definitions.

Let X be the set of all software systems, which can be implemented. The set is huge and unknown; however, it exists and is finite (the number of atoms in the universe is finite; hence, the number of computer memory states is also finite). We do not give any precise definition of software systems (set X); however, we believe that software developers can agree on what a software system is. In other words, X is a primitive notion motivated only informally, by an appeal to intuition and everyday experience. The other primitive notions are architectural decision issues and architectural decision alternatives. We borrow the notation from [11] and denote the set of architectural decision issues by *ADIssue* and the set of architectural decision alternatives by *ADAlternative*.

Let $p \in ADIssue$ be an architectural decision issue. The set of all possible solutions (alternatives) to issue p is denoted $A(p)$ where $A(p) \subseteq ADAlternative$. Making an architectural decision on issue p means selecting an alternative a from $A(p)$. The selected alternative becomes the outcome of the decision on issue p. Thus, the set of all architectural decision outcomes is a subset of the set of alternatives, $ADOutcome \subseteq ADAlternative$.

An example of an architectural decision issue p can be the architectural pattern of a software system. The set $A(p)$ of possible solutions to issue p comprises layers, pipe and filter architecture, service component architecture, etc. Each of those solutions is an alternative of a decision on issue p. Another example of an issue is the protocol used in communication between the major system components. The set of possible solutions to this issue comprises REST, SOAP and some other protocols. If we select REST, then the meaning of this selection is the set of all software systems that use REST at the component level. If we select another alternative, e.g., SOAP, then we look at quite another set of systems. A taxonomy of architectural decision issues that refer to SOA systems has been defined in [11].

The proposed semantics of the architectural decision model defines the meaning of architectural decision alternatives, the meaning of outcomes of architectural decisions and the relations between issues and alternatives in the set X of software systems.

Let $p \in ADIssue$ be an architectural decision issue. The outcome of issue p in system x, i.e., the alternative selected and implemented in system x, is denoted $x(p)$.

Definition 1. *Systems $x_1, x_2 \in X$ are architecturally equivalent with respect to issue p if and only if $x_1(p) = x_2(p)$.*

Architectural equivalence introduced in Definition 1 is a relation $arch_p \subseteq X \times X$, such that $(x_1, x_2 \in arch_p) \Leftrightarrow [x_1(p) = x_2(p)]$

Lemma 1. *Architectural equivalence $arch_p$ of software systems with respect to issue p is an equivalence relation.*

Proof. The thesis is true, because the identity relation (=) used to compare the alternatives of issue p in Definition 1 is an equivalence relation on *ADAlternative*. □

The set of equivalence classes of relation $arch_p$ forms a partition of X. Any two software systems of an equivalence class are architecturally equivalent with respect to issue p. Therefore, we can identify an equivalence class of relation $arch_p$ with software architecture, and the quotient set $Arch_p = X / arch_p$ with the set of all software architectures, with respect to issue p. The equivalence class of a system x with respect to issue p is denoted $[x]_p$.

Definition 2. *The semantics of an alternative $a \in A(p)$ is a set $S(a) \subseteq X$ such that $x(p) = a$ for each system $x \in S(a)$.*

Lemma 2. *Semantics $S(a)$ of alternative a is an equivalence class of relation $arch_p$ on X.*

Proof. Let $x \in X$ be a system such that $x(p) = a$. We will show that $S(a) = [x]_p$. Assume $x_1 \in S(a)$. Then, $x_1(p) = a$ according to Definition 2. Hence, $x_1, x \in arch_p$ according to Definition 1 and $x_1 \in [x]_p$. Now, assume $x_1 \in [x]_p$. If this is the case, then $x_1(p) = a$ according to Definition 1 and $x_1 \in S(a)$ according to Definition 2. □

The relation of architectural equivalence of software systems with respect to an issue p can easily be extended into a relation with respect to a set of issues.

Let $ap \subseteq ADIssues$ be a set of architectural decision issues.

Definition 3. *Systems $x_1, x_2 \in X$ are architecturally equivalent with respect to a set of issues ap if and only if $x_1, x_2 \in arch_p$ for each $p \in ap$.*

Architectural equivalence introduced in Definition 3 is a relation $arch_{ap} \subseteq X \times X$ such that $(x_1, x_2 \in arch_{ap}) \Leftrightarrow (\forall\ p \in ap) [x_1, x_2 \in arch_p]$

Lemma 3. *Architectural equivalence $arch_{ap} \subseteq X \times X$ of systems with respect to a set of issues ap is an equivalence relation.*

Proof. The thesis is true, because the relation $arch_p$ used to compare software systems in Definition 3 is an equivalence relation on X. □

The set of equivalence classes of relation $arch_{ap}$ forms a partition of X. Any two software systems of an $arch_{ap}$ equivalence class are architecturally equivalent with respect to all issues of the set *ap*. Therefore, we can look at an equivalence class of relation $arch_{ap}$ as a software architecture with respect to a set of issues *ap*. The quotient set $Arch_{ap} = X / arch_{ap}$ is the set of software architectures, which are different with respect to architectural decision issues of *ap*. The equivalence class of a system x with respect to a set of issues *ap* is denoted $[x]_{ap}$.

Software design process comprises a set of architectural decisions, each of which narrows the set of feasible software systems down. At the beginning, before the first decision has been made, the solution space is equal to set X of all software systems. When the first decision a_{p1} on issue p_1 is made, the solution space reduces to $S (a_{p1})$. The next issue p_2 is considered within the scope of $S (a_{p1})$. When the design continues and decision a_{p2} on issue p_2 is made, the solution space reduces down to $S (a_{p1}) \cap S (a_{p2})$. This way, the designer decides among alternatives of subsequent issues and the outcomes of those decisions constitute the design.

Definition 4. *A design is a partial function d: ADIssue → ADAlternative, which defines the outcomes of architectural decision issues.*

Function d is partial, because not all issues of *ADIssue* need to be resolved. For example, if the *implementation method* in Figure 1 is *Java*, then the issue of *BPEL version* need not be considered. The domain of function d is denoted $Dom (d)$, and the range is denoted $Ran (d)$ where $Dom (d) \subseteq ADIssue$ and $Ran (d) \subseteq ADAlternative$.

Definition 5. *The semantics of a design d: ADIssue → ADAlternative is a set $S (d)$ of software systems such that $S (d) = \bigcap_{p \in Dom(d)} S (d (p))$.*

Example 1. To exemplify the definition of design *d*, consider a process of defining a high level architecture of a software system. The process can start with issue p_1 of choosing the general architectural pattern of the system. Assume that the selected alternative a_{p1} is service component architecture. This decision reduces the solution space to set $S (a_{p1})$ of systems that comply with service component architectural pattern. Now, one can decide on issue p_2 of choosing the implementation technology. Decision a_{p2} to select Java (Figure 1), reduces the solution space to service component architecture implemented in Java. Then, issue p_3 of selecting the communication protocol can be considered. Assume that the outcome a_{p3} of this decision is to select SOAP. The resulting design d is a function from design issues *ADIssue* to possible solutions *ADAlternative* such that:

- *d* (*architectural pattern*) = *service component architecture*
- *d* (*implementation technology*) = *Java*
- *d* (*communication protocol*) = *SOAP*

The semantics of design *d* is a set of software systems that comply with *service component architecture*, are implemented in *Java* and use *SOAP* for the component communication.

Design *d* defines a set of issues related to the organization of software components, together with a set of solutions recommended to those issues. The semantics of design *d* is the set of software systems that solve the

issues of $Dom(d)$ in the same way, i.e., the set of systems that have the same architecture with respect to issues of $Dom(d)$.

Let d: $ADIssue \rightarrow ADAlternative$ be an arbitrary design.

Theorem 1. *Semantics $S(d)$ of a design d is an equivalence class of relation $arch_{Dom(d)}$ on X.*

Proof. Let $x \in X$ be a software system such that $x(p) = d(p)$ for each $p \in Dom(d)$. We will show that $S(d) = [x]_{Dom(d)}$. Assume $x_1 \in S(d)$. Then, $x_1 \in S(d(p))$ for each $p \in Dom(d)$, according to Definition 5, and $x_1(p) = d(p)$ for each $p \in Dom(d)$, according to Definition 2. Hence, $x_1, x \in arch_p$ for each $p \in Dom(d)$, according to Definition 1, and $x_1, x \in arch_{Dom(d)}$ according to Definition 3. Now, assume $x_1 \in [x]_{Dom(d)}$. If this is the case, then $x_1, x \in arch_p$ for each $p \in Dom(d)$, according to Definition 3, $x_1(p) = x(p) = d(p)$ for each $p \in Dom(d)$, according to Definition 1, and $x_1 \in S(d)$ according to Definition 5. ☐

4. Architectural Decision Model

The architectural alternatives define pieces of the software architecture, which have been designed to solve a variety of architectural decision issues. Some of those pieces can work together, while the others cannot. Some of those pieces must be applied together. The ability to work together is expressed by a set of relations: *isCompatibleWith*, *isIncompatibleWith* and *forces*, which are defined between architectural decision alternatives in [11]. A conceptually similar, yet different, set of relations: *Constrains*, *Forbids* and *ConflictsWith* are defined between decisions in [6]. No such or similar relations are defined in [5].

Architectural alternatives and the outcomes of architectural decisions influence the set of architectural decision issues, which have to be solved in the future. The influence is partially captured by relations *Enables* and *Comprises* in [6], *Depends on* and *Refines* in [5], and *triggers* and *decomposesInto* in [11]. The relations defined by different authors are conceptually similar, but a precise comparison of their meaning is difficult, because of the lack of formally defined semantics. In all cases, however, these relations introduce partial ordering of architectural decisions that are made in a software design process.

The model of architectural decisions is a graph, the nodes of which are issues and alternatives, and edges are relations defined on issues and alternatives. An example graph showing a subset of issues that must be resolved when implementing components in Service Component Architecture (SCA) is shown in Fig. 1. The meaning of the architectural decision model is defined in [5,6,11] only informally by textual explanations and illustrating diagrams. In this Section we propose a formal definition of the model semantics, which includes architectural issues, alternatives and a set of relations.

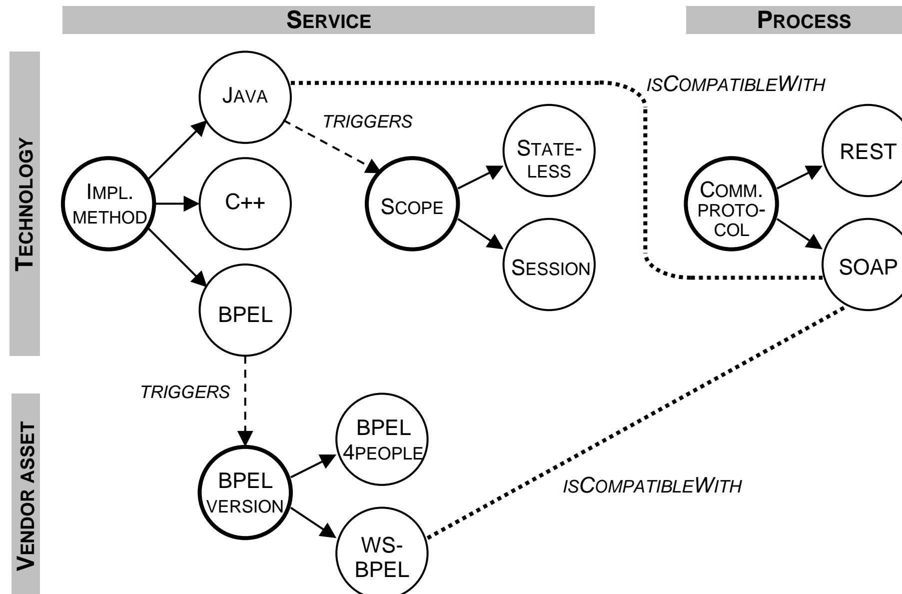

Fig. 1. Sample architectural decision model for a service component implementation. Issues: *Implementation method*, *Scope* (of a Java module), *BPEL version* (for a BPEL implementation), *Communication protocol*. Alternatives: all the other circles.

4.1. *Logical Relations between Alternatives*

Let $p_1, p_2 \in ADIssue$ be arbitrary architectural decision issues and $a_1, a_2 \in ADAlternative$ be arbitrary alternatives in those issues, such that $a_1 \in A(p_1)$, $a_2 \in A(p_2)$. The following relations between the alternatives are defined.

Definition 6. *Alternative a_1 is compatible with a_2, if their semantics are overlapping:*
$isCompatibleWith(a_1, a_2) \Leftrightarrow [S(a_1) \cap S(a_2) \neq \phi]$.

Definition 7. *Alternative a_1 is incompatible with a_2, if their semantics are disjoint:*
$isIncompatibleWith(a_1, a_2) \Leftrightarrow [S(a_1) \cap S(a_2) = \phi]$.

Definition 8. *Assume that $p_1 \neq p_2$ and $a_1 \neq a_2$. Alternative a_1 forces a_2, if a_1 is compatible with a_2 and is incompatible with any other alternative in p_2:*
$forces(a_1, a_2) \Leftrightarrow$
$[isCompatibleWith(a_1, a_2) \wedge (\forall a_i \in A(p_2): a_i \neq a_2)\, isIncompatibleWith(a_1, a_i)]$.

It can easily be seen that relation *isCompatibleWith* is symmetric and reflexive, while *isIncompatibleWith* is symmetric and irreflexive. The relations are complementary, i.e., any two alternatives are either compatible or incompatible, but not both. An alternative can *force* only a compatible one.

Similar relations, with the same names: *isCompatibleWith*, *isIncompatibleWith* and *forces* have been introduced in [11] at an intuitive level, without giving any formal semantics. This makes a comparison of those definitions with the above ones difficult, yet not impossible. One difference is that the three relations are assumed mutually exclusive and exhaustive in [11], while according to our definitions, only two relations *isCompatibleWith* and *isIncompatibleWith* are complementary; relation *forces* is a subset of *isCompatibleWith*. The reasoning behind our definitions is twofold. First, any two alternatives can either work together or not, therefore the choice between *isCompatibleWith* and *isIncompatibleWith* is mutually exclusive and exhaustive. Second, choosing an alternative can force to choose another alternative, only if both of them can work together. Therefore, these alternatives must be compatible.

Another difference between the definitions relates to the properties of relation *isCompatibleWith*. It is claimed in [11] that the relation is transitive. To verify this claim, let we look at an example.

Example 2. Figure 1 shows an architectural decision model for an SCA service component implementation. Java components can implement SOAP protocol, hence, alternatives: *Java* and *SOAP* are compatible. BPEL components can also implement SOAP, therefore, alternatives: *WS-BPEL* and *SOAP* are also compatible. But either we implement the component in Java or in BPEL. Alternatives: *Java* and *WS-BPEL* are not compatible. This shows that *isCompatibleWith* relation is not transitive.

Relation *isCompatibleWith* plays a role in the software design process. It is not transitive; hence, it is not an equivalence relation and cannot be used to partition the model into equivalence classes. However, this relation can be used to isolate consistent slices of a design, which are composed of alternatives that can all work together. Let $Alt = \{a_1, \ldots a_n\}$ be an arbitrary set of alternatives to a set of architectural decision issues $\{p_1, \ldots p_n\}$, $a_i \in A(p_i)$. The semantics of Alt is set $S(Alt)$ of software systems: $S(Alt) = \bigcap_{a \in Alt} S(a)$.

Definition 9. *Set of alternatives $Alt \subseteq ADAlternative$ is consistent, if $S(Alt) \neq \phi$.*

Lemma 4. *If $Alt = \{a_1, \ldots a_n\}$ is a consistent set of alternatives, then $isCompatibleWith(a_i, a_j)$, for all $i, j \leq n$.*

Proof. Assume $isIncompatibleWith(a_i, a_j)$, $i, j \leq n$, for an arbitrary pair of alternatives from Alt. Then, $S(a_i) \cap S(a_j) = \phi$ by definition 7, which leads to $S(A) = \phi$. □

Design d is feasible, i.e., $S(d) \neq \phi$, if $Ran(d)$ is a consistent set of alternatives. This means, according to the above lemma, that any two outcomes of a feasible design $d(p_i)$, $d(p_j)$ are compatible for each $p_i, p_j \in Dom(d)$.

4.2. *Decision Dependencies*

Architectural decisions made on design issues are the basic building blocks of a software design process. The set of issues, which have to be solved, is not constant throughout the design, but new issues can be created by decisions that have been made earlier. This dependency is covered by *triggers* relation between alternatives and issues.

Definition 10. *The domain of an architectural decision issue* $p \in ADIssue$ *is set* $S(p) \subseteq X$, *such that* $S(p) = \bigcup_{a \in A(p)} S(a)$.

Let $p_1, p_2 \in ADIssue$ be arbitrary architectural decision issues, $a \in A(p_1)$ be an alternative in issue p_1, and $a_1, \ldots a_k \in A(p_2)$ be the set of alternatives in issue p_2.

Definition 11. *Alternative a triggers issue* p_2, *if the domain of* p_2 *includes the semantics of a, and alternative a is compatible with all the alternatives in* p_2:
$triggers(a, p_2) \Leftrightarrow [\, S(a) \subseteq S(p_2) \wedge \forall a_i \in A(p_2)\ isCompatibleWith(a, a_i)\,]$.

The rationale behind this definition is the following. Inclusion $S(a) \subseteq S(p_2)$ means that after selecting alternative a, the design must be implemented within the domain of issue p_2; if this was not the case, then the system could be implemented without deciding on p_2, within $S(a) - S(p_2)$. Compatibility condition means that the set of software systems $S(a)$ defined by alternative a can be partitioned into a family of smaller sets $S(a_1), \ldots S(a_k)$.

A complex issue p can be decomposed into a set of simpler issues tha reside at the same architectural level, i.e., conceptual, technology or vendor asset. Relation *decomposesInto* expresses functional aggregation of concerns, and allows splitting a complex issue into a set of issues, each of which addresses a distinct aspect of software [11]. The solution of the complex issue is an intersection of solutions to the aggregated issues.

Let $p, p_1, \ldots p_n \in ADIssue$ be a set of architectural decision issues and let $a_p, a_{p1}, \ldots a_{pn} \in ADOutcome$ be the outcomes of decisions on those issues.

Definition 12. *Issue p decomposes into issues* $p_1, \ldots p_n$, *if the semantics of each alternative a in p is equal to the semantics of a consistent set* Alt_a *of alternatives* $a_1, \ldots a_n$ *in* $p_1, \ldots p_n$, respectively:
$decomposes(p, p_1, \ldots p_n) \Leftrightarrow$
$(\forall a \in A(p))(\exists Alt_a = \{a_1, \ldots a_n\})[(\forall i \le n)(a_i \in A(p_i)) \wedge (S(a) = \bigcap_{i=1..n} S(a_i)]$.

Lemma 5. *Relation decomposesInto is transitive.*

Proof. The relation is defined using set intersection, which is a transitive operation. ☐

The formula in Definition 12 is valid also for the outcomes. If one decides on issue p and selects the outcome $a_p \in A(p)$, then the set $Alt_{ap} = \{a_{p1}, \ldots a_{pn}\}$ defines the outcomes of issues $p_1, \ldots p_n$, i.e.: $S(a_p) = \bigcap_{i=1..n} S(a_{pi})$.

5. Metamodel for architectural design decisions

Software architecture is a result from decisions made throughout the long term process of software design and evolution. This process consists of decisions that are made, and then potentially changed in response to changes in the business environment. If we want to preserve the past experience, we should model not only the current software architecture but also its change over time. This can be done by the appropriate software tools.

A domain metamodel for capturing architectural decisions is shown in Figure 2. It uses UML classes to introduce the three basic entities defined in Section 3: *ADIssue*, *ADAlternative* and *ADOutcome*. The attributes of those entities are similar to the ones introduced in [11]. An instance of *ADIssue* describes a single architectural decision issue. The *name* attribute gives identity to each issue and *scope* identifies the design models affected by the issue. The *problemStatement* attribute defines the issue itself. Additional explanatory information on the issue is pointed to by *backgroundReading*. The *decisionDrivers* attribute captures forces that influence the decision on the architectural design issue [15]. As the system evolves, new issues can arise and new instances of *ADIssue* can be added. A modified issue is viewed as a new one; therefore no attributes of type

modified... are present. The *status* tells the architect whether the issue is *open*, i.e., has to be solved, *solved*, or *closed*, i.e., withdrawn and perhaps substituted by a new one. The meaning of *decomposesInto* association is given in Definition 12.

An instance of *ADAlternative* describes a single alternative in an architectural decision issue. The *description* attribute defines the alternative, i.e., gives a possible solution to the architectural decision issue. References to additional explanatory information on the alternative, points in favor and points against the alternative are listed in *backgroundReading*, *pros* and *cons*, respectively. The links between an issue and the alternatives in this issue are given by *isAlternativeIn* association. Each alternative is linked to exactly one issue, while an issue may have many alternative solutions. The meaning of *triggers* association is given in Definition 11.

Instances of *ADOutcome* represent decisions on issues. Any such decision is linked to an issue by *isOutcomeOf* association, and to the chosen alternative in this issue by *chooses* association. Multiplicities "many-to-one " of both associations indicate that decisions may change over time in the course of the software evolution process. When a new decision on an issue is made, a new instance of *ADOutcome* is added, but no instances are removed. This way, all the past design knowledge and experience is preserved and can be reused if needed. If a modification to the system architecture is considered, the designer can easily find all the previous decisions related to the issue at hand, together with the justification of their choice or removal.

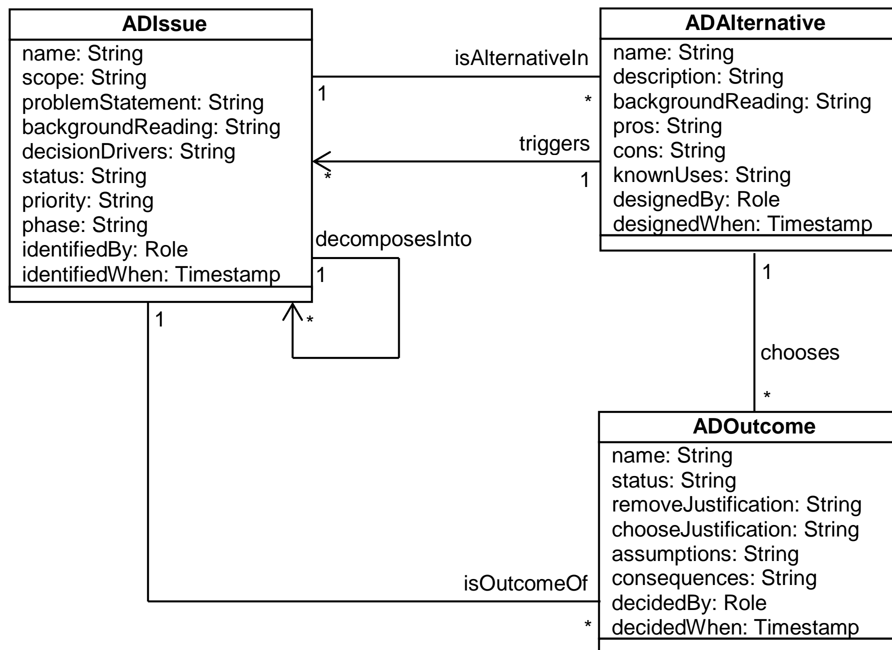

Fig. 2. UML metamodel for architectural decision modelling.

The current architecture of software is defined by the set of all the current outcomes of the architectural decisions issues. The set of the current outcomes define the design function *d* introduced in Definition 4. The domain of function *d* consists of issues linked by *isOutcomeOf* and the range of *d* consists of alternatives linked by *chooses* association. The order of making architectural decisions in a software project should comply with the direction of both relations *triggers* and *decomposesInto*.

The metamodel described in this section can be used as a basis for building the supporting software tools. In fact, several such tools have already been built. For example, a tool [18] to visualize architectural design decision model [6], Archium tool [19] to support the model introduced in [5], Architectural Decision Knowledge Wiki [20] to present the model introduced in [11]. The tools mentioned above are capable to store, retain and present the architectural knowledge during software design and evolution. The properties of the architectural decision model proved in this paper can enable the next generation tools to verify the model consistency.

6. Conclusions

In this paper, we presented an architectural decision model based on the work described by others in the literature [5,6,10,11,15], and defined a formal semantics for the basic concepts and relationships that exist in the architectural knowledge. The formalization can help in understanding the meaning of architectural decisions and the relations that exist between the decision elements, i.e., issues, alternatives, and outcomes. The definition of architectural decision semantics is general and does not relate to any particular type of the architecture, e.g. layered or SOA.

The decision model introduced in Section 4 may serve as a basis for implementing a supporting software tool. Aiming at this goal, we defined in Section 5 an UML-based metamodel, capable of capturing the currently valid architectural decisions as well as the history of decision changes. The properties of the architectural decision model proved in Sections 3 and 4 can be used by the tool as rules to follow in verifying the model consistency.

Future work concerns aspects related to the evolution of SOA systems. Although issues located in different architectural layers of a SOA system (business process, services and service components) influence each other, the need for evolution originates in the business. A business processes can be specified using Business Process Modelling and Notation (BPMN), and implemented as a network of communicating services. A need to change the current business process may be compromised by the existing configuration of services. These constrains can be expressed by additional relations. We are working on a transformational approach to business process evolution and implementation in SOA [24].

References

[1] IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, *IEEE 1471*, 2007.

[2] Systems and software engineering – Architecture description, *ISO/IEC/IEEE 42010*, 2011.

[3] P. Kruchten, Architectural Blueprints – The "4+1" View Model of Software Architecture, *IEEE Software* 12 (6) (1995) 42–50.

[4] J. Tyree and A. Akerman, Architecture decisions: Demistyfying architecture, *IEEE Software* 22 (2) (2005) 19–27.

[5] A. Jansen and J. Bosch, Software Architecture as a Set of Architectural Design Decisions, in *Proc. 5th Working IEEE/IFIP Conf. on Software Architecture*, IEEE Computer Society, 2005, pp. 109–120.

[6] P. Kruchten, P. Lago and H. van Vliet, Building up and reasoning about architectural knowledge, in *Proc. 2nd Int. Conf. on the Quality of Software Architectures* (QoSA 2006), LNCS 4214, Springer, Heidelberg, 2006, pp. 43–58.

[7] M.A. Babar, I. Gorton and B. Kitchenham, A framework for supporting architecture knowledge and rationale management, in *Rationale Management in Software Engineering*, eds. A.H. Dutoit, R. McCall, I. Mistrik and B. Paech, (Springer, 2006), pp. 237–254.

[8] R. Capilla, F. Nava and J.C. Dueñas, Modeling and Documenting the Evolution of Architectural Design Decisions, in *Proc. 2nd Workshop on Sharing and Reusing Architectural Knowledge – Architecture, Rationale, and Design Intent*, IEEEXplore, 2007.

[9] A. Tang, Y. Jin and J. Han, A rationale-based architecture model for design traceability and reasoning, *J. Syst. Software* 80 (6) (2007) 918–934.

[10] R.C. de Boer, R. Farenhorst, P. Lago, H. van Vliet, V. Clerc and A. Jansen, Architectural Knowledge: Getting to the Core, in *Proc. 3rd Int. Conf. on the Quality of Software Architectures* (QoSA 2007), LNCS 4880, Springer, Heidelberg, 2008, pp. 197–214.

[11] O. Zimmermann, J. Koehler, F. Leymann, R. Polley and N. Schuster, Managing architectural decision models with dependency relations, integrity constraints, and production rules, *J. Syst. Software* 82 (8) (2009) 1249–1267.

[12] A. Jansen, P. Avgeriou and J. van der Ven, Enriching Software Architecture Documentation, *J. Syst. Software* 82 (8) (2009) 1232–1248.

[13] N.B. Harrison, P. Avgeriou and U. Zdun, Using Patterns to Capture Architectural Decisions, *IEEE Software* 24 (4) (2007) pp. 38–45.

[14] M.A. Babar, T. Dingsoyr, P. Lago and H. van Vliet (eds.), *Software Architecture Knowledge Management: Theory and Practice* (Springer, Berlin, 2009.

[15] U. van Heesch, P. Avgeriou and R. Hilliard, Forces on Architecture Decisions – A Viewpoint, in *Proc. Joint 10th Working IEEE/IFIP Conf. on Software Architecture & 6th European Conf. on Software Architecture*, IEEE Computer Society, 2012, pp. 101–110.

[16] R. Capilla, O. Zimmermann, U. Zdun, P. Avgeriou and J.M. Küster, An Enhanced Architectural Knowledge Metamodel Linking Architectural Design Decisions to other Artifacts in the Software Engineering Lifecycle, in *Proc. 5th European Conf. on Software Architecture* (ECSA 2011), LNCS 6903, Springer, Heidelberg, 2011, pp. 303–318.

[17] A. Zalewski, S. Kijas and D. Sokołowska, Capturing Architecture Evolution with Maps of Architectural Decisions 2.0, in *Proc. 5th European Conf. on Software Architecture* (ECSA 2011), LNCS 6903, Springer, Heidelberg, 2011, pp. 83–96.

[18] L. Lee and P. Kruchten, A Tool to Visualize Architectural Design Decisions, in *Proc. 4th International Conference on Quality of Software-Architectures: Models and Architectures* (QoSA 2008), LNCS 5281, Springer, Heidelberg, 2008, pp. 359-362.

[19] A. Jansen, J.S. van der Ven, P. Avgeriou and D.K. Hammer, Tool Support for Architectural Decisions, in *Proc. 6th IEEE/IFIP Working Conference on Software Architecture* (WICSA 2007), IEEE Computer Society, 2007, pp. 44-53.

[20] N. Schuster and O. Zimmermann, Architectural Decision Knowledge Wiki, IBM alphaWorks, March 2008, http://www.alphaworks.ibm.com/tech/adkwik.

[21] M. Shahin, P. Liang and M.R. Khayyambashi, Architectural Design Decision: Existing Models and Tools, in *Proc. Joint 8th Working IEEE/IFIP Conf. on Software Architecture & 3rd European Conf. on Software Architecture*, IEEE Computer Society, 2009, pp. 293–296.

[22] A. Tang, P. Avgeriou, A. Jansen, R. Capilla and M.A. Babar, A comparative study of architecture knowledge management tools, *J. Syst. Software* 83 (3) (2010) 352–370.

[23] R. Weinreich and I. Groher, A Fresh Look at Codification Approaches for SAKM: A Systematic Literature Review, in *Proc. 8th European Conference on Software Architecture* (ECSA 2014), LNCS 8627, Springer, Heidelberg, 2014, pp. 1–16.

[24] A. Ratkowski, K. Sacha and A. Zalewski, Optimization of Business Processes in Service Oriented Architecture, in *Proc. 2012 IEEE Enterprise Distributed Object Computing Conf. Workshops*, IEEE Computer Society, 2012, pp. 42–50.