Mapping Architectural Decay Instances to Dependency Models

Ran Mo*, Joshua Garcia†, Yuanfang Cai*, Nenad Medvidovic†

*Computer Science Department
Drexel University,
Philadelphia, USA
{yc349, rm859}@drexel.edu

†Computer Science Department
University of Southern California,
Los Angeles, USA
{joshuaga, neno}@usc.edu

***Abstract*—The architectures of software systems tend to drift or erode as they are maintained and evolved. These systems often develop *architectural decay instances*, which are instances of design decisions that negatively impact a system's lifecycle properties and are the analog to code-level decay instances that are potential targets for refactoring. While code-level decay instances are based on source-level constructs, architectural decay instances are based on higher levels of abstractions, such as components and connectors, and related concepts, such as concerns. Unlike code-level decay instances, architectural decay usually has more significant consequences. Not being able to detect or address architectural decay in time incurs *architecture debt* that may result in a higher penalty in terms of quality and maintainability (interest) over time. To facilitate architecture debt detection, in this paper, we demonstrate the possibility of transforming architectural models and concerns into an *extended augmented constraint network (EACN)*, which can uniformly model the constraints among design decisions and environmental conditions. From an ACN, a *pairwise-dependency relation (PWDR)* can be derived, which, in turn, can be used to automatically and uniformly detect architectural decay instances.**

I. Introduction

It has been widely documented that engineers tend to make decisions about system changes without careful consideration of the impact of those changes on the system's architecture. As a result, the architecture of a software system will eventually deviate from the original intent, resulting in *architectural drift* and *erosion* [20]. In turn, this makes new modifications to a system increasingly time consuming and costly, while the system's reliability decreases. Such a system incurs a *technical debt* [5], where short-term compromises lead to significant long-term problems in terms of reduced ability of fixing bugs or adding new features.

A system suffering from architectural drift or erosion will eventually develop a number of *decay instances* (sometimes referred to as "bad smells"), many of which will be at the level of the system's architecture. For example, a system component may lose its conceptual coherence (becoming a "jack of many trades") or it may interact with other components using multiple, disparate software connectors [16]. Such decay instances are instances of architectural design decisions that negatively impact system lifecycle properties, such as understandability, testability, extensibility, and reusability. They are architecture-level analogs to the better known code decay instances [10] and are thus targets for restructuring. These architecture decay instances usually have more significant consequences, and we consider them as *architecture debt* that deserves special attention.

Unlike *code decay instances*, which have been widely studied, the nature of *architectural decay instances* is not nearly as well defined and understood. While the two types of decay instances may occur simultaneously, the architectural decay instances are likely to have a more significant impact on the system, yet they may not manifest themselves in (a corresponding set of) code decay instances. Relying on our experience in the area of software systems architecture, on published literature, and on available systems (e.g., open-source), we recently proposed a categorization of architectural decay instances [13], such as component envy, connector envy, and scattered functionality, which are based on standard architecture building blocks: components, connectors, interfaces, and configurations.

Although these definitions are counterparts of code decay instances, identifying those architectural decay instances in a system is more challenging. Unlike code decay instances that are defined based on source code constructs, and thus can be detected against source code, the definitions and thus detection of architectural decay instances rely on a higher level of abstraction. For example, in order to detect component envy in source code, one has to determine which source-level element belongs to which components. Moreover, some architectural decay-instance definitions involve *concerns*, a concept beyond the basic architecture definitions. For example, the architecture decay instance, *Scattered Functionality*, involves multiple components that are responsible for realizing the same high-level concern and, some of those components are responsible for orthogonal concerns. Without a unified representations of components, connectors, and concerns, it is difficult to automatically detect these decay instances.

Based on the perspective that a software system's architecture is the set of principal design decisions governing a system [20], in this paper, we propose to transform architectural models and related concepts, such as concerns, into an *augmented constraint network* (ACN) [7], which was designed to uniformly model the constraints among both design decisions and environmental conditions using a constraint network.

978-1-4673-6443-0/13/$31.00 © 2013 IEEE

MTD 2013, San Francisco, CA, USA

In this paper, we show how the original ACN model can be extended to model the relation between components and concerns. We call the extended ACN an EACN.

We show that it is possible to map components, connectors, interfaces and concerns, as well as their relations, into an EACN. Using this mapping, it becomes possible to model all the defined architectural decay instances using EACN concepts, such as *pairwise-dependency relations* (PWDRs) that can be derived from an EACN. The implication of the mapping is that, once we represent all the architectural elements as design decisions and their constraints, these architectural decay instances can be modeled as patterns in the dependency model created by the constraint network, which, in turn, can be automatically and uniformly detected.

II. Definitions and Background

In this section, we provide the definition of basic architectural concepts and ACN modeling, as the background of our work.

A. Architectural Concept for Decay-Instance Relevant View

Our definitions of architectural concepts are not intended to be complete; they are restricted to those architectural concepts that will be useful for identifying decay instances.

A software system's architecture is a graph *G* whose vertices are "bricks" (software components and connectors) and whose topology represents the interconnections among those bricks. In order to represent and detect architectural decay instances, we model a system's architecture as a tuple comprising *G*, the nonempty set of "words" *W* that are used to "describe" (i.e., implement) the system modeled by the architecture, and the nonempty set of "topics" *T* addressed by the system; each topic is defined as a probability distribution over the system's words. By examining the words that have the highest probabilities in a topic, the meaning of that topic can be discerned. In this way, a topic can serve as a representation of a concern addressed by a software system. In other words, the set of topics *T* is a representation of the system's concerns.

$A = (G, W, T)$
$G = (B, L)$
$W = \{w_i \mid i \in \mathbb{N}\}$
$T = \{z_i \mid i \in \mathbb{N}\}$
$z = Pd(W)$

A brick *B* can be either simple or composite. A composite brick *CB* is an architecture in its own right, allowing for multiple levels of architectural abstraction. We omit the formal definition of *CB* for brevity; the definition is essentially the same as that for architecture *A* above. Each simple brick *SB* is a tuple comprising the brick's internal state *S*, its interface $\mathbb{I}$, set of operations $\mathbb{O}$, the map *M* that relates the operations and the interfaces through which they are exported, and the probability distribution θ over the system's topics *T*.

$B = SB \cup CB$
$SB = \{b_i \mid i \in \mathbb{N}\}$
$b = (S, \mathbb{I}, \mathbb{O}, M, \theta_b)$

A brick's state *S* is defined as a set of variables *var*, where each variable is a tuple comprising a name *n* (which must be one of the words in *W*), a type *t*, and a value *val*.

$S = \{v_i \mid i \in \mathbb{N}\}$
$v = (n, t, val)$
$n \subseteq W$

A brick's interface consists of a set of interface elements ie, each of which is a tuple comprising a name *ni* (which must be one of the words in *W*), a possibly empty set of parameters P, and a possibly empty set of return variables RV.

$\mathbb{I} = \{ie_i \mid i \in \mathbb{N}\}$
$ie = (ni, P, RV)$
$ni \subseteq W$
$P = \{v_j \mid j \in \mathbb{N}_0\}$
$RV = \{v_k \mid k \in \mathbb{N}_0\}$

A brick's operation op is a tuple comprising a set VO of variables that comprise the operation's state, an algorithm alg that realizes the operation, a probability distribution θ_{op} over the operation's topics (called "document-topic distribution" for short) and a function op_type. T_{op} are the set of topics over which θ_{op} is distributed, i.e., T_{op} represents the operation's concerns. op_type determines whether a topic in T_{op} is application-specific (pertaining to the system's "business logic") or application-independent (pertaining to the bricks' interaction needs).

$\mathbb{O} = \{op_i \mid i \in \mathbb{N}\}$
$op = (VO, alg, \theta_{op}, op\_type)$
$VO = \{v_l \mid l \in \mathbb{N}_0\}$
$\theta_{op} = Pd(T_{op})$
$T_{op} = \{z_j \mid j \in \mathbb{N}\}$
$op\_type : T_{op} \rightarrow SP$
$SP = \{spec, indep\}$

The mapping relation M relates a brick's operations with the interface elements through which they are accessed. The tuples in the relation are restricted such that every interface is paired with an operation in the tuple if their types match. Note that multiple operations can be part of different tuples comprising the same interface.

$M = \{(ie_k, op_j) \mid \forall ie_k \in \mathbb{I} \mid \exists op_j \in \mathbb{O} \mid \forall v_m \in ie_k.P \cup ie_k.RV \mid \exists v_h \in op_j.VO \mid v_h.t = v_m.t\}$

The document-topic distribution θ_b is a probability distribution over topics T. θ_b represents the extent to which the concerns represented by topics T are present within the brick b.

$\theta_b = Pd(T)$

A link l is a tuple comprising a source interface src and a destination interface dst. Links are the channels over which

components and connectors transfer data and control over their interfaces.

$L = \{l_i \mid i \in \mathbb{N}_0\}$
$l = (src, dst)$
$src, dst \in \mathbb{I}$

A component c is a brick whose interfaces are all application-specific and whose topics are primarily application-specific. Interfaces are considered application-specific if the words naming the interface are application-specific. In particular, each word of the system is classified as either application-independent or application-specific. A brick is considered primarily application-specific if, for each topic that occurs in the component with a probability above a threshold th_z_c, that topic is application-specific. th_z_c is specified by an architect.

$C = \{c_i \mid c_i \in B \wedge i \in \mathbb{N} \wedge c_i.\mathbb{I} = I \wedge \forall z_c \in T \mid P(z_c \mid c_i) > th\_z_c \Rightarrow z\_type(z_c) = spec\}$
$I = \{ia_j \mid j \in \mathbb{N} \wedge ia_j \in \mathbb{I} \wedge ia_j.ni \subset AS\}$
$AS = \{w_j \mid j \in \mathbb{N} \wedge w_j \in W \wedge w\_type(w_j) = spec\}$
$w\_type : W \rightarrow SP$
$0 \leq th\_z_c \leq 1$
$z\_type : T \rightarrow SP$

A connector n is a brick whose interfaces are all application-independent. Interfaces are considered application-independent if the words naming the interface are application-independent. A brick is considered primarily application-independent if, for each topic that occurs in the connector with a probability above a threshold th_z_n, that topic is application-independent. th_z_n is specified by an architect. $TP_n(n)$ is a relation indicating the types a connector n may be.

$N = \{n_i \mid n_i \in B \wedge i \in \mathbb{N} \wedge n_i.\mathbb{I} = D \wedge \forall z_n \in T \mid P(z_n \mid c_i) > th\_z_n \Rightarrow z\_type(z_n) = indep\}$
$D = \{id_j \mid j \in \mathbb{N} \wedge id_j \in \mathbb{I} \wedge id_j.ni \subset AD\}$
$AD = \{w_k \mid k \in \mathbb{N} \wedge w_k \in W \wedge w\_type(w_k) = indep\}$
$0 \leq th\_z_n \leq 1$
$TP_n(n) = \{ty_i \mid i \in \mathbb{N} \wedge n \in N \wedge ty_i \in \{proc\_call, event, stream, distributor, data\_access, adaptor, arbitrator\}$

B. Augmented Constraint Network

Our previous work on *augmented constraint networks* (ACN) [8], [9], [22] makes it possible to capture both architectural decisions and concerns in a formal, unified way. An ACN consists of a *constraint network* (CN), a *dominance relation* (DR), and a *cluster set* (CS). That is: $ACN = \langle CN, DR, CS \rangle$. The core computation model of an ACN is the CN [14]. A CN consists of a set of *variables*, V, with their *domains*, D, and the constraints among these variables, C: $CN = \langle V, D, C \rangle$. The *variables* can model either design dimensions, such as classes or algorithms, or relevant concerns, such as business rules or features. The *domain* of a *variable* comprises a set of *values*, each representing a possible choice within that dimension. Figure 1 shows a CN modeling the design of a simplified graph library, which has three variables (line 1-3), each having a domain with two values, and three constraints (line 4-7). The first variable, $density$, models graph density required by the user. ds and alg model data structure and algorithm choices. As an example, line 4 is a constraint modeling the fact that the decision to use an adjacency matrix data structure assumes that the client needs dense graphs.

$1 : density : (dense, sparse);$
$2 : ds : (matrix, list);$
$3 : alg : (matrix\_alg, list\_alg);$
$4 : ds = matrix => density = dense;$
$5 : ds = list => density = sparse;$
$6 : alg = array\_alg => ds = array;$
$7 : alg = list\_alg => ds = list;$

Fig. 1. The Constraint Network for a Simple Graph Library

The dominance relation $DR \subseteq V \times V$, that augments a CN, models an asymmetric relation among decisions, formalizing the concept of *design rule* [3]. For example, the pair $(ds, density)$ belongs to the DR that augments the sample CN, indicating that the decisions about which data structure to use cannot influence the client's requirement on the density of graphs to be used.

Another augmentation, CS, models the concept of *module*, which is another essential concept in software design that a CN does not lend itself to modeling: different stakeholders can view the same design in different ways. For example: we can cluster all connectors into one module, and all components into another module, with each component as a sub-module; or, we can cluster all concern-related elements into a concern module, and all GUI-related elements into a GUI module. We model the multiple *modularization candidates* using a *cluster set* (CS) that consists of a set of $clusterings$. Each $clustering$ expresses *a priori* aggregation of subsets of variables into candidate modules. The CS of an ACN contains multiple $clusterings$, reflecting different stakeholders' views of the design.

Maintaining satisfiability through minimal perturbation of an ACN forms the concept of *pairwise dependence relation* (PWDR) [9]: $PWDR \subseteq V \times V$. If $(u, v) \in PWDR$, meaning that v depends on u: when u is changed and the consistency of the constraint network is broken, v must be changed in some minimal way to restore the consistency to the constraint network. Given a PWDR relation derived from an ACN and a $clustering$ of it, a *design structure matrix* (DSM) can be automatically derived.

A DSM is a square matrix in which columns and rows are labeled with the same set of design variables in the same order. Each $clustering$ of an ACN is an ordered tree structure that can be used to determine the DSM variable order. A marked cell of the DSM can represent the dependency relation among the variables. If $(u, v) \in PWDR$, then the cell in row v, column u will be marked. The blocks along the diagonal can be used to model $modules$ within a $clustering$. For example,

Figure 3 shows a DSM that visualizes a $clustering$ with two top-level modules, the first module has 10 variables, and the second module has 17 variables that are further clustered into 15 sub-modules. In the next section, we introduce how this ACN model can be extended to capture architectural entities, concerns and decay instances.

III. The ACN Mapping of Architectural Models

Both architectural elements and concerns can be uniformly represented as variables of an ACN. The dependencies among architectural elements can be formalized as logical constraints, as we did in our prior work [21], [2], [17], [6]. We propose to use a new data structure, called CE, to represent the relation between architectural elements and their concerns. CE is defined as a tuple: $\langle crn, ele, p\rangle$, where crn models a set of concerns, ele models architectural building blocks, and p models the probability that ele is involved in crn.

For example, if we identified that the $ResourceManagement$ component is involved in the $PersonnelResources$ concern with a probability of 48%, then $\langle PersonnelResources, ResourceManagement, 0.48\rangle$ would be a member of CE. As a result, the extended ACN, which we call $EACN$, is defined as: $EACN = \langle CN, DR, CS, CE\rangle$.

Next, we introduce the mapping between the architectural model and the ACN model, as well as how the mapping can be used to detect architectural decay instances, which can be visualized in a DSM. For the sake of space, we restrict our discussion to four of the architectural decay instances we have defined. After that, we use an example to show how architectural decay instances can be visualized base on these mappings.

A. Architecture Model and Decay-Instance Mappings

First of all, we model each element in an architecture model using a design variable. For example, we model a component using a variable c, model an operation of this component as $c.op$, and model one of its interfaces $c.i$. Similarly, we can model a concern as crn. Without knowing the concrete decisions of these variables, following prior work on ACN modeling [7], we can generally model that they all have a domain with at least two values: $(orig, other)$, where $orig$ means a current decision and $other$ means some other decision that is different from the current one.

We also generally model the relation between these variables using assumption relations expressed as logical constraints. For example, either an operation op uses or implement an interface i, the decision of op has to make assumptions on i. Accordingly, we can model this assumption relation as: $op = orig \implies i = orig$. Based on this constraint, our tools will compute that (op, i) is a pair that belongs to PWDR.

If concerns and components are reverse-engineered from source code, then we can use the CE relation in the extended EACN model to express this relation. For example, if a component c is related to concern crn with a probability of 67%, then we model this as a tuple $(crn, c, 67\%)$, which belong to the CE relation of the EACN.

Based on this mapping, we now introduce how these architectural decay instances can be mapped as patterns within these ACN data structures, so that they can be detected automatically.

Connector Envy. Components with *Connector Envy* encompass extensive interaction-related functionality that should be delegated to a connector. Formally, a component $c \in COMP$, where $COMP = C$, suffers from *Connector Envy* in the following cases:

- *Connector Interface Implementation*. In this case, a component exposes an application-independent interface: Interface ia of component c exhibits this decay instance if:

  $\exists op \in c.\mathbb{O} \mid ia \in c.\mathbb{I} \wedge (ia, op) \in c.M \wedge op\_type(op) = indep.$

  In ACN, the operation and interface can be modeled as op and ia. To distinguish application-specific and application-independent operations, we can create a clustering where these two types of operations are separated into two sets app_spec and app_indep. Now this decay instance can be defined as: there exists interface ia and the operation op of a component c. If op depends on ia, that is, the pair (op, ia) belongs to $PWDR$, and op is in the app_indep cluster, then the Connector Envy decay instance occurs. Formally:

  $\exists ia \in c.I \wedge \exists op \in c.O \mid (op, ia) \in PWDR \wedge op\_type(op) \in app\_indep$

- *Unacceptably High Connector Concern*. In this case, a single application-independent concern as represented by a topic is too high as specified through a threshold selected by an architect. A component $c \in COMP$ exhibits this decay instance case if:

  $\exists z \in T \mid z\_type(z) = indep \wedge P(z \mid c) > th\_z_n.$

  Suppose there is a clustering in the ACN cluster set, CS, where application-independent concerns and application-dependent concerns are separated into two sets, crn_indep and crn_dep, then this decay instance can be expressed as follows: there exists concern z and a component c, where $z \in crn\_indep$. If there is a tuple (z, c, pr) in the CE of the EACN, where $pr > th\_z_n$, then this smell occurs. Formally:

  $\exists z \in T \wedge \exists c \in COMP \mid (z, c, pr) \in CE \wedge z \in crn\_indep \wedge pr > th\_z_n$

- *Data Flow Interface Envy*. For this decay instance, a component's interface simply passes the parameters of the interface as the return value of some other interface. Component interfaces $i_1, i_2 \in c.I$ exhibit this decay instance if:

$\exists p_i \in i_1.P, \exists rv_i \in i_2.RV \mid i_1 \neq i_2 \wedge p_i = rv_i$

In ACN, this decay instance can be modeled as follows: There exists interfaces i_1, i_2 of a component. If for any pairs of i_1's parameter (modeled using variable $c.1.p$) and i_2's return value (modeled using variable $c.i_2.rv$), $c.i_2.rv$ depends on $c.i_1.p$, then the Connector Envy decay instance occurs. Formally:

$\exists c \in COMP \wedge \exists i_1 \in c.I \wedge \exists i_2 \in c.I - i_1 \mid (c.i_2.rv, c.i_1.p) \in PWDR$

Scattered Parasitic Functionality. Scattered Parasitic Functionality describes a system where multiple components are responsible for realizing the same high-level concern and, additionally, some of those components are responsible for orthogonal concerns. This decay instance violates the principle of separation of concerns in two ways. First, this decay instance scatters a single concern across multiple components. Secondly, at least one component addresses multiple orthogonal concerns. In other words, the scattered concern infects a component with another orthogonal concern, akin to a parasite.

Formally, components $c_1, c_2 \in COMP$ are part of the *Scattered Parasitic Functionality decay instance* if:

$\exists z_1 \in T \mid P(z_1 \mid c_1) > th_1 \wedge \exists z_2 \in T \mid P(z_2 \mid c_2) > th_1 \wedge z_1 = z_2 \wedge \exists z_3 \in T \mid P(z_3 \mid c_2) > th_2 \wedge z_1 \neq z_3$, where th_1, th_2 are proportions such that $0 \leq th_1 \leq 1$ and $0 \leq th_2 \leq 1$. th_1 and th_2 specify the acceptable degree of scattering per topic.

In EACN, this decay instance can be expressed as follows: there exists concerns z_1, z_2, z_3, and components c_1, c_2. If the CE of the EACN contains the following tuples (z_1, c_1, p_1), (z_2, c_2, p_2) and (z_3, c_2, p_3), where $z_1 = z_2$ and $z_1 \neq z_3$, and $p_1 > th_1$, $p_2 > th_1$, and $p_3 > th_2$, then the Scattered Parasite Functionality decay instance occurs. Formally:

$\exists z_1, z_2, z_3 \in T \wedge \exists c_1, c_2 \in COMP \mid (z_1, c_1, p_1) \in CE \wedge (z_2, c_2, p_2) \in CE \wedge (z_3, c_2, p_3) \in CE \wedge p_1 > th_1 \wedge p_2 > th_1 \wedge p_3 > th_2 \wedge z_1 = z_2 \wedge z_1 \neq z_3$

Extraneous Adjacent Connectors. The *Extraneous Adjacent Connector* decay instance occurs when two connectors of different types are used to link a pair of components. Components $c_1, c_2 \in COMP$ and connectors $n_1, n_2 \in CONN$, where $CONN = N$, are involved in an instance of an *Extraneous Adjacent Connector* decay instance if:

$connected(c_1, n_1) \wedge connected(n_1, c_2) \wedge connected(c_1, n_2) \wedge connected(n_2, c_2) \wedge TP_n(n_1) \neq TP_n(n_2)$

In EACN, this decay instance can be modeled as follows: there exists a pair of components c_1 and c_2, and two connectors, n_1 and n_2, of different types. If both components depend on both connectors, then the Extraneous Adjacent Connector decay instance occurs. Formally:

$\exists c_1, c_2 \in COMP \wedge \exists n_1, n_2 \in CONN \mid (c_1, n_1) \in PWDR \wedge (c_2, n_1) \in PWDR \wedge (c_1, n_2) \in PWDR \wedge (c_2, n_2) \in PWDR \wedge TP_n(n_1) \neq TP_n(n_2)$

Brick Concern Overload. Brick Concern Overload is inspired by Stal's architectural decay instance called *Component Responsibility Overload* [18]. While Stal uses the word "responsibility," we use the synonymous word "concern." Furthermore, instead of just letting the decay instance apply to components, we allow it to apply to both components and connectors, i.e., bricks.

Formally, a brick has concern overload if it handles an excessive number of concerns. A brick $b \in B$ suffers from this decay instance if:
$|\{z_j \mid j \in \mathbb{N} \wedge P(z_j|b) > th\_z_b\}| > th_t$, where th_z_b is the threshold indicating that the topic is high for the brick and $0 \leq th\_z_b \leq 1$. th_t is a threshold indicating the acceptable number of concerns per brick and $th_t \in \mathbb{N}$.

Using EACN, this decay instance can be expressed as follows: there exists a brick b and concerns z_i, where $i \in \mathbb{N}$. There are multiple tuples in the CE of the ACN that relates b with different concerns, with probabilities greater than th_z_b. If the number of such tuples is greater than th_t then this decay instance occurs. Formally:

$\exists b \in B \wedge i \in \mathbb{N} \wedge \exists z_i \in T \wedge \exists p > th\_z_b$ such that $BC = \{(z_i, b, p) \in CE\} \wedge |BC| > th_t$

B. An Example

As shown in the previous section, all the architectural decay instances can be modeled using PDWRs, which in turn, can be visualized in a DSM. To illustrate how a DSM can be used to visualize architectural decay instances, we will use an application for distributed deployment of personnel in operations involving emergencies, such as natural disasters, search-and-rescue efforts, and military crises. The system is called the Emergency Response System (ERS) and was introduced in [15]. In ERS, A small number of laptop computers oversee an operation and primarily interact with a set of higher-end handhelds. These handhelds are in charge of specific segments of the operation and interact with a large number of lower-end handhelds, which are used by first-line responders.

ERS is designed using the C2 architectural style [19] and implemented in Java using the PrismMW middleware platform [15]. The C2 style organizes components and connectors in a layered fashion. Communication between them occurs solely through message passing. Therefore, the messages (also referred to as events) are the major data elements of ERS.

A simple instantiation of the ERS architecture is depicted in Figure 2. ERS's *Map* component maintains a model of the system's overall resources: regions of space, personnel,

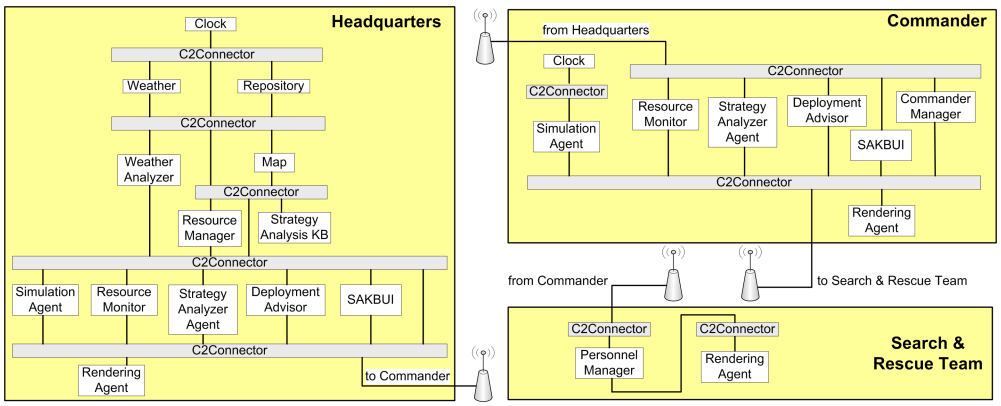

Fig. 2. A view of ERS's architecture deployed on three hardware hosts. White rectangles are the system's components and gray rectangles are the connectors.

emergency vehicles, and buildings. These resources are permanently stored inside the *Repository*. *StrategyAnalyzerAgent*, *DeploymentAdvisor*, and *SimulationAgent* components, respectively, analyze the deployments of personnel with respect to the areas that require emergency response, suggest deployments of personnel based on their availability as well as the positions of buildings and emergency vehicles, and incrementally simulate the outcome of deploying personnel to different areas with different emergency requirements. *StrategyAnalysisKB* and *SAKBUI* components store the strategy rules and provide the user interface for changing these rules, respectively. *ResourceManager*, *CommanderManager*, *PersonnelManager*, and *ResourceMonitor* components enable the allocation and transfer of resources and periodically update the state of resources. *Weather* and *WeatherAnalyzer* components provide weather information and analyze the effects of weather conditions. Finally, the *RenderingAgent*s provides the user interface of the application.

Figure 3 shows a DSM for the ERS system. The first block contains the 10 concerns recovered from the ERS source code. The next block contains the system's architectural building blocks. The number in a cell is the probability of the component in that row being involved in the concern in that column. These numbers are automatically calculated using Latent Dirichlet Allocation (LDA) [4], which is a statistical language model used in information retrieval. We automate distinguishing between application-specific and application-independent concerns using a machine learning-based technique described in [12].

In the ERS example, all four decay instances we have identified to date can be visualized in the DSM, which can be automatically derived from an EACN: the cells with the dark background and white font represent the dependencies that indicate architectural decay instances. For example, the SimulationAgent and the DeploymentAdvisor components both depend on ResourceMonitor and both components depend on $ComponentInterface$, which in turn, depends on $ConnectorInterface$, showing the symptoms of the extraneous adjacent connector decay instance, according to its formal definition. The dependencies of the SimulationAgent and DeploymentAdvisor on the ResourceMonitor are caused by the fact that the SimulationAgent and DeploymentAdvisor directly call the event `handle` method of ResourceMonitor. The DSM also shows that all components depend on a shared $DataInterface$, i.e., $Event$, to send or receive data, exhibiting the ambiguous interface symptom since that DataInterface is a key part of the ambiguous interface provided by PrismMW. The ResourceMonitor and StrategyAnalyzerAgent depend on both ComponentInterface and ConnectorInterface, exhibiting the connector envy decay instance since the dependencies on these interfaces imply that these two components implement functionality of both components and connectors.

The DSM also shows that concerns modeled by variables 3, 6, 7, and 9 have a strong scattered functionality decay instance because the number of components they involve is larger than three, which is the median number of components involved in a concern in the ERS system and is used as a threshold in this case. Thresholds can be determined manually or automatically by using statistics (such as the median, mean plus standard deviation, etc.). Concern 3 is about $Event\ and\ Message\ Management$. Concern 6 is about $Weather$. Concern 7 is about functionality of the $Commander\ and\ Agents$. Concern 9 is about $Usage\ of\ Shared\ Data\ Structures$. After further inspection, it turns out that concern 3 has the scattered functionality decay instance because all components communicate through messages—an example where the decay instance is not necessarily a problem with the system's design. Concern 9 also is a strong decay instance, but in this case it is an indication that the design can be improved to better separate concerns.

IV. Conclusion

Architectural decay instances, that is, instances of architecture debt, negatively impact the lifecycle properties of a software system and affect their major architectural elements,

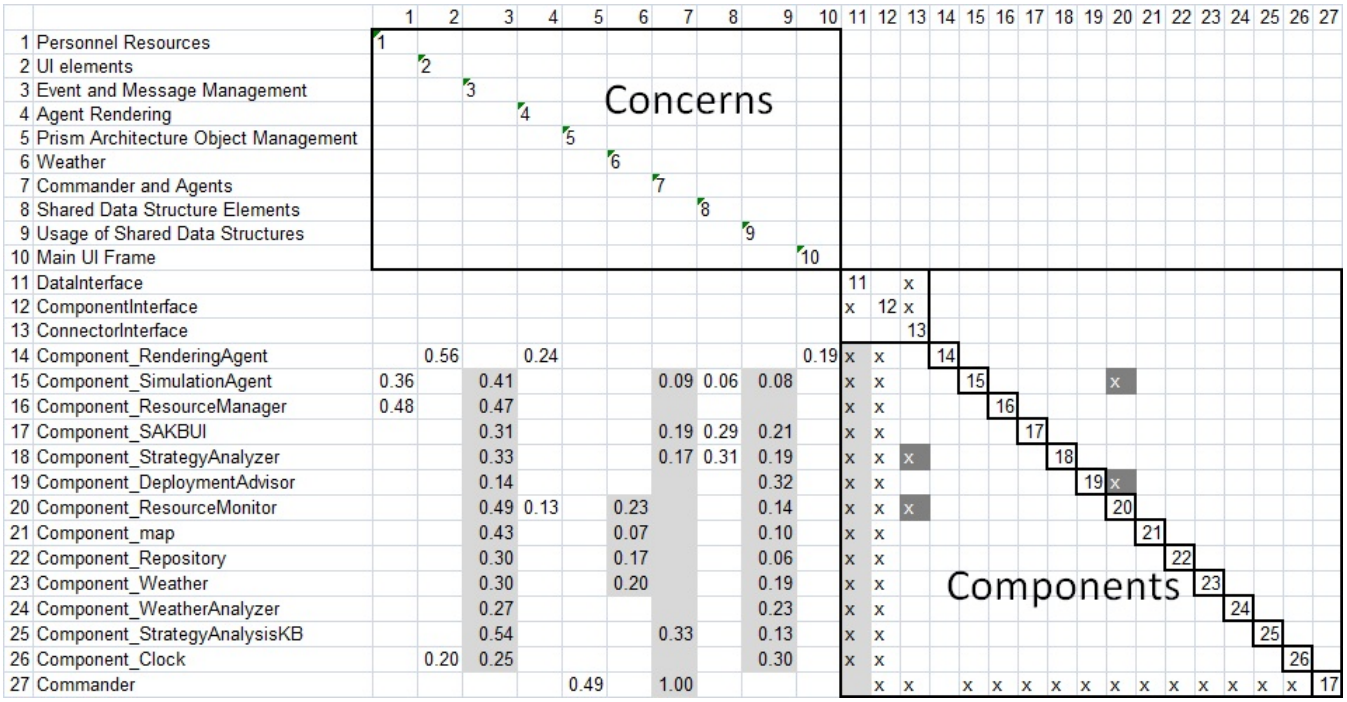

Fig. 3. The design structure matrix visualizing the ERS architecture, concerns, and decay instances

such as components and connectors. Thus, they are also targets for restructurings that would ameliorate those negative impacts. However, before restructuring can be performed, architectural decay instances must be detected. In this paper, we have provided uniform models that enable the detection of architectural decay instances. In particular, we have demonstrated that architectural decay instances (sometimes referred to as architectural "bad smells") can be modeled as EACNs and PWDRs that can be derived from EACNs. We also have shown how architectural decay instances can be manifested in DSMs, which is a visualization of the PWDR, CE and clustering of an ACN.

For our future work, we are currently using ACNs to help detect an expanded list of architectural decay instances on a variety of software systems for which we have reliable representations of their architectures. In particular, the architectures of the systems include those recovered with the aid of those systems' architects and developers [11], e.g., Apache Hadoop, a widely used open-source framework for distributed processing of large datasets across clusters of computers [1].

While the architectures of these systems are derived from code, we also plan to explore the use of prevailing UML models in order to transform them to DSMs so that we can detect architectural decay instances using those models. Lastly, we also plan to detect and locate architectural decay instances by comparing how architectural building blocks *should* change together based on the architectural structure and how they *actually* change together as reflected in the revision history. The rationale is that if, for example, two separate components always change together to accommodate modification requests but they belong to two separate concerns that are supposed to evolve independently, we consider these components to be exhibiting an architectural decay instance.

Acknowledgments

This work was supported in part by the National Science Foundation under grants CCF-0916891, CCF-1065189, CCF-1116980 and CCF-1117593.

References

[1] (2012) Poweredby - Hadoop Wiki. [Online]. Available: http://wiki.apache.org/hadoop/PoweredBy

[2] A. Avritzer, D. Paulish, and Y. Cai, "Coordination implications of software architecture in a global software development project," in *Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*, ser. WICSA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 107–116. [Online]. Available: http://dx.doi.org/10.1109/WICSA.2008.16

[3] C. Y. Baldwin and K. B. Clark, "Design Rules, Vol. 1: The Power of Modularity," 2000.

[4] D. Blei, A. Ng, and M. Jordan, "Latent dirichlet allocation," *The Journal of Machine Learning Research*, vol. 3, pp. 993–1022, 2003.

[5] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, and N. Zazworka, "Managing technical debt in software-reliant systems," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, ser. FoSER '10. New York, NY, USA: ACM, 2010, pp. 47–52. [Online]. Available: http://doi.acm.org.libproxy.usc.edu/10.1145/1882362.1882373

[6] Y. Cai, S. Huynh, and T. Xie, "A framework and tool supports for testing modularity of software design," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ser. ASE '07. New York, NY, USA: ACM, 2007, pp. 441–444. [Online]. Available: http://doi.acm.org/10.1145/1321631.1321704

[7] Y. Cai and K. Sullivan, "A formal model for automated software modularity and evolvability analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 21, no. 4, pp. 21:1–21:29, Feb. 2013. [Online]. Available: http://doi.acm.org/10.1145/2377656.2377658

[8] Y. Cai and K. J. Sullivan, "Simon: modeling and analysis of design space structures," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ser. ASE '05. New York, NY, USA: ACM, 2005, pp. 329–332. [Online]. Available: http://doi.acm.org/10.1145/1101908.1101962

[9] ——, "Modularity analysis of logical design models," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 91–102. [Online]. Available: http://dx.doi.org/10.1109/ASE.2006.53

[10] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[11] J. Garcia, I. Krka, C. Mattmann, and N. Medvidovic, "Obtaining ground-truth software architectures," in *International Conference on Software Engineering*, 2013, to Appear.

[12] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai, "Enhancing architectural recovery using concerns," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 552–555. [Online]. Available: http://dx.doi.org.libproxy.usc.edu/10.1109/ASE.2011.6100123

[13] G. E. Joshua Garcia, Daniel Popescu and N. Medvidovic, "Identifying Architectural Bad Smells," in *13th European Conference on Software Maintenance and Reengineering*, 2009.

[14] A. K. Mackworth, "Consistency in networks of relations," *Artificial intelligence*, vol. 8, no. 1, pp. 99–118, 1977.

[15] S. Malek, M. Mikic-Rakic, and N. Medvidovic, "A style-aware architectural middleware for resource-constrained, distributed systems," *IEEE Transactions on Software Engineering*, pp. 256–272, 2005.

[16] N. R. Mehta, N. Medvidovic, and S. Phadke, "Towards a taxonomy of software connectors," in *Proc. of the 22nd International Conference on Software Engineering*, 2000.

[17] K. Sethi, Y. Cai, S. Wong, A. Garcia, and C. Sant'Anna, "From retrospect to prospect: Assessing modularity and stability from software architecture," in *Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*. IEEE, 2009, pp. 269–272.

[18] M. Stal, "The beauty and quality of software," http://http://stal.blogspot.com/2011/01/beauty-and-quality-of-software.html.

[19] R. Taylor, N. Medvidovic, K. Anderson, E. Whitehead Jr, J. Robbins, K. Nies, P. Oreizy, and D. Dubrow, "A component-and message-based architectural style for GUI software," *Software Engineering, IEEE Transactions on*, vol. 22, no. 6, pp. 390–406, 2002.

[20] R. Taylor, N. Medvidovic, and E. Dashofy, "Software Architecture: Foundations, Theory, and Practice," 2009.

[21] S. Wong and Y. Cai, "Improving the efficiency of dependency analysis in logical decision models," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 173–184. [Online]. Available: http://dx.doi.org/10.1109/ASE.2009.55

[22] S. Wong, Y. Cai, G. Valetto, G. Simeonov, and K. Sethi, "Design rule hierarchies and parallelism in software development tasks," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 197–208. [Online]. Available: http://dx.doi.org/10.1109/ASE.2009.53