Coupling Software Architecture and Human Architecture for Collaboration-Aware System Adaptation

Christoph Dorn[*][†]
[*] Distributed Systems Group
Vienna University of Technology, Austria
dorn@infosys.tuwien.ac.at

Richard N. Taylor[†]
[†] Institute for Software Research
University of California, Irvine, USA
{cdorn | taylor}@uci.edu

***Abstract*—The emergence of socio-technical systems characterized by significant user collaboration poses a new challenge for system adaptation. People are no longer just the "users" of a system but an integral part. Traditional self-adaptation mechanisms, however, consider only the software system and remain unaware of the ramifications arising from collaboration interdependencies. By neglecting collective user behavior, an adaptation mechanism is unfit to appropriately adapt to evolution of user activities, consider side-effects on collaborations during the adaptation process, or anticipate negative consequence upon reconfiguration completion.**

Inspired by existing architecture-centric system adaptation approaches, we propose linking the runtime software architecture to the human collaboration topology. We introduce a mapping mechanism and corresponding framework that enables a system adaptation manager to reason upon the effect of software-level changes on human interactions and vice versa. We outline the integration of the human architecture in the adaptation process and demonstrate the benefit of our approach in a case study.

***Index Terms*—collaboration topology, software architecture, runtime mapping, architecture reconfiguration, dynamic adaptation**

I. Introduction

In 2006, Northrop et al. [1] identified Ultra-Large-Scale (ULS) systems as the major future software engineering challenge. Among the defining characteristics of ULS systems are decentralized control, conflicting and changing requirements, continuous evolution, heterogeneous and dynamic system elements, ubiquitous failures, and erosion of the people/system boundary. This paper focuses primarily on the implication of that last aspect on system (self-) adaptation. People are no longer just the "users" of a system but an integral part [1] p13. Consequently human interactions are highly relevant to the design and adaptation of ULS systems ([1] p31ff). We believe that this is true not only for ULS systems but also for traditional medium and large-scale systems. Any system heavily relying upon significant user collaboration needs to explicitly address human interaction implications during design-time and runtime.

Among the many adaptation approaches, architecture-driven techniques appear to be the most applicable to systems exhibiting ULS characteristics. Kramer and Magee [2] argue that an architecture-based approach provides (i) concepts and principles applicable across domains, (ii) sufficient abstraction from the algorithmic and network level while still capturing dynamic change, and (iii) scalability through hierarchical composition, thereby facilitating the specification of systems of systems. In addition, architecture-driven adaptation techniques are among the earliest [3] and continuously relevant approaches [4] as demonstrated by successful application to mobile environments [5], robotics systems [6], and adaptive service compositions [7].

Current architecture-driven adaptation mechanisms, however, consider only the software system and remain unaware of the ramifications arising from collaboration interdependencies. A system neglecting the collective user behavior might suffer from some of the following example weaknesses:

- The system is unable to support the efficient operation and evolution of user behavior. For example, failing to provide appropriate coordination mechanisms when groups of users change their behavior from sequential resource access to simultaneous resource access.
- Conversely, the system cannot anticipate the consequences of particular software adaptations. Disregarding, for example, user proximity, user role, or user capacity might result in reconfigurations that jeopardize a team's performance due to increasing the likelihood of information overload, information delay, information scarcity, or resource access conflicts.
- Likewise, the system is unable to reason about side-effects during the software reconfiguration process. A database schema update, for example, might have the implicit assumption that humans are in a state of quiescence upon commencing an update, potentially interrupting all ongoing interactions.
- The system remains unaware of users becoming bottlenecks. Unavailable or overloaded users slow down critical processes when they are responsible for manually triggering key tasks.

We propose linking the system's software architecture to human interactions. Specifically we describe the system's users in terms of human components and collaboration connectors along with their means of communication and coordination. To this end, we apply the human Architecture Description Language (hADL) introduced in our previous work [8] for specifying a system's underlying collaboration topology, and the eXtensible Architecture Description Language (xADL [9]) for specifying the software architecture. Explicit non-trivial design-time mappings between hADL and xADL elements allow, during runtime, the matching of software component (and connector) instances to users and their interactions. Adaptation rules can subsequently utilize the hADL model, for example, for prioritizing the replication of components associated with key collaborators.

The main contributions of this paper are

- a model for mapping from software architecture to human collaboration patterns and vice versa
- a framework for detecting runtime software architecture changes and reflecting those changes in the human collaboration topology according to the predefined mappings.
- a discussion on integrating human architecture and software architecture for system adaptation.
- a case study demonstrating the benefit of turning software architecture-centric self-adaptation strategies to be collaboration-aware.

The remainder of this paper is structured as follows. Section II and III provide a motivation scenario and a discussion of related work, respectively. Section IV summarizes background information, an overview of our approach, and the architecture mapping rationale. Section V details the design-time mapping specification and the runtime mapping process. We discuss the application of our framework for collaboration-aware system self-adaptation in Section VI. A case study in Section VII demonstrates actual adaptation benefits. Finally, Section VIII gives an outlook on future work and concludes this paper.

II. Motivating Scenario

Monitoring and safety systems range in scope from a small security team handling an office building to thousands of personnel in back offices and on site at geographically distributed locations to secure critical infrastructure. These systems tightly interweave people and software components and hence need co-adaptation of collaboration structures and software architectures. In the building monitoring case, back office operators utilize high definition video streams, floor plans, building sensor feeds, occupancy logs, and communication channels with on-site security staff. Reassigning observation tasks among team members, reacting to non-responding team members, or adding new team members are examples of collaboration-driven adaptations that result in changes to the underlying software structure.

The adaptation mechanism needs to react to software-level events such as failing components, congested data links, and emergence of new information sources. At the same time it requires maintenance of various QoS metrics such as acceptable video delay, video stream availability, and bandwidth cost through continuous adaptation of video relay replication and video stream rerouting.

In the presence of scarce resources, the adaptation mechanism has to prioritize the adaptation of particular relays and video streams. To this end, it requires awareness of the collaboration topology and user roles. Consider the software architecture in Figure 1 consisting of components for StreamingServers, VideoSources, GUIs for each role, and connectors for coordinating video publishing, subscribing, and delivering activities. This architecture may serve as the underlying communication infrastructure for two, quite distinct collaboration topologies (Fig. 2 and Fig. 3). The publish/subscribe human architecture in Figure 2 specifies the following human components: *FieldAgents* provide video streams (*PubStreams*), whereas *Backoffice Agents*, *Assistants*, and *Team leaders* subscribe to video streams (*SubStreams*). *VideoPubSub* collaboration connectors—typically but not necessarily implemented as software entities—manage video stream publication and subscription. Video feeds may be replicated across multiple VideoPubSub connectors in accordance with the software architecture. All users have access to a *WallScreen* (a collaboration object of type Shared Artifact) for displaying relevant video streams. The collaboration topology in Figure 3 lacks such a flat organizational hierarchy and instead features a pipes/filters-style collaboration structure. Individual agents receive their video feeds as deemed relevant by their predecessor. A Backoffice Agent, for example, routes a *PipeStream* to an Assistant. Ultimately only the Team leader has access to the WallScreen.

Suppose an adaptation mechanism reconfigures the software architecture to maintain system reliability by avoiding individual StreamingServers from becoming overloaded. Simultaneously, it should ensure that the team leader has (the most) reliable streams. Without a mapping between software and collaboration structure, it would be unable to make an informed decision between adaptation action *"replicate team leader video streams"* (suitable for the human architecture in Fig. 2) or action *"equal component replication along the video relay chain"* (suitable for the human architecture in Fig. 3). We will be using these configurations throughout the paper for explaining the mapping process at runtime and design-time, the adaptation process, and the final evaluation.

III. Related Work

Our work builds on the insights of architecture-based adaptation research. As early as 1999, Orzeiy et al. [3] outlined the process for reflecting runtime changes in an architectural model as the basis for dynamic adaptation. Subsequent work focused predominately on architecture-based adaptation techniques such as the Rainbow framework [10], the K-Component Architecture Meta-model [11], Model-based development [12], or Object-oriented design adaptation [13]. In line with such previous work, our framework also features an architecture runtime manager and the adaptation mechanism follows the feedback loop described by the autonomic

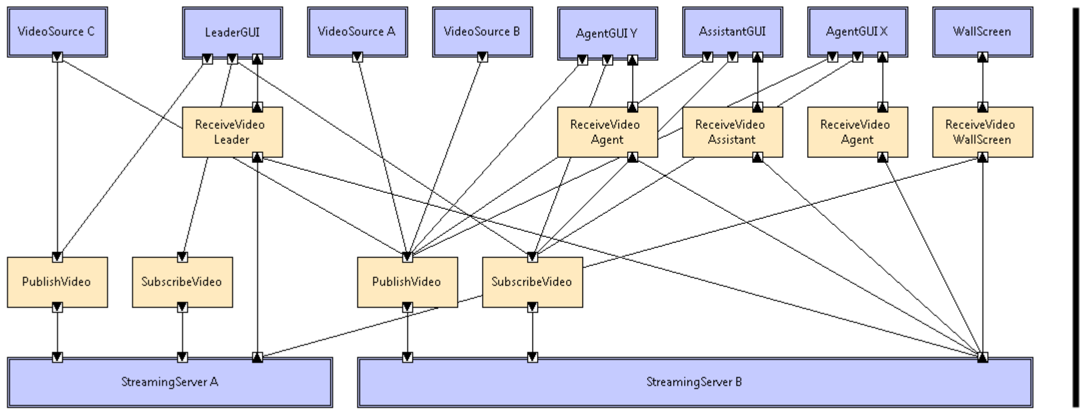

Fig. 1. Software Architecture: Surveillance Video Monitoring.

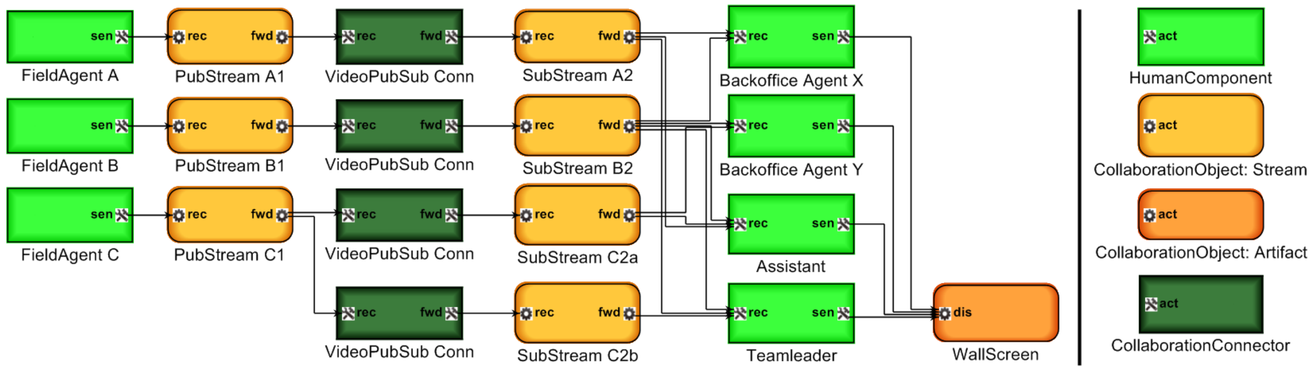

Fig. 2. Collaboration Architecture: Publish/Subscribe-style Surveillance Team. Information flows from left to right along collaboration links. Each link connects two collaboration actions (3-letter abbreviated: *sen*d, *f*or*w*ar*d*, *rec*eive, *dis*play).

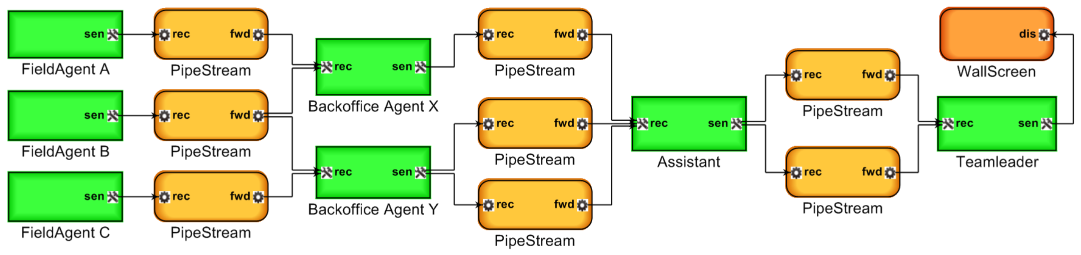

Fig. 3. Collaboration Architecture: Pipes/Filters-style Surveillance Team. Information flows from left to right along collaboration links. Each link connects two collaboration actions (3-letter abbreviated: *sen*d, *f*or*w*ar*d*, *rec*eive, *dis*play).

computing MAPE-K model: Monitoring, Analysis, Planning, Execution, and Knowledge.

As we pointed out in the introduction, these techniques focus exclusively on adapting the software architecture. User preferences and user context drive adaptation in mobile scenarios (e.g., the MADAM architecture model [5]) but the applied techniques still remain unaware of collaboration dependencies. The novel aspect of our research is mapping the human architecture (hADL) to the software architecture (xADL) at designtime and runtime. As we will demonstrate in Section VI, having two distinct, but constantly synchronized views on the system gives rise to unique adaptation opportunities.

Note that linking of the xADL and hADL architecture models should not be mistaken for the three-layer architecture model [14], typically applied for self-adaptation in the robotics domain [6]. The three-layer model describes a strict hierarchical separation of goal management, change management, and change execution. In contrast, we propose to apply software architecture and human architecture simultaneously across all steps of the MAPE-K model.

Enhancing software architecture models with domain specific properties enables analysis beyond structural consistency. Edwards and Medvidovic [15] apply multi-model composition in their XTEAM framework to simulate reliability, power consumption, and performance. Di Ruscio et al. [16] utilize model mapping and transformation techniques for integrating multiple architecture concerns (e.g., fault tolerance and activity flow). The SASSY framework [17] provides service activity schemas and service sequence scenarios to specify QoS requirements in service-oriented architectures. Finally, Bhave et al. [18] augment software architectures with physical properties and behavioral annotations, thus enabling an integrated specification of cyber-physical systems such as quadrotors. The main difference compared to our approach is the extremely tight coupling of the various architectural views such that no separate mapping and tracing is foreseen or required during runtime.

The business process modeling domain traditionally included some aspects of human involvement. Business Process Model and Notation BPMN [19] consists of constructs for describing activities in business processes, their dependencies, artifacts, and involved events. BPMN processes typically map to BPEL, the Business Process Execution Language, for execution. The BPEL4People [20] extension utilizes human tasks for integrating users into otherwise Web-service based workflows. Human tasks support assignment to generic roles, ownership delegation, and coordination mechanisms such as *four eyes*, *nomination*, or *escalation*. Both languages primarily target service-oriented architectures with limited or no support for other common architectural styles such as Peer-to-Peer, Components and Connectors (C2), or Publish-Subscribe. Likewise, support for collaboration is limited to isolated execution of individual task items from a work list. Dynamic patterns for joint work on shared artifacts, publish-subscribe information distribution, organizational control, or request routing in social networks and thus also the patterns' adaptation implications [21] remain outside the scope of BPMN and BPEL. The Human-provided Service framework (HpS) [22] offers more flexible user collaboration but lacks support for structural patterns at the human level and the software level.

As a final note for clarification and caution: we cannot rely on insights from Conway's Law [23] or socio/technical congruence [24] when describing the mapping between collaboration structure and software architecture. We model the structure of the users' organization rather than the developers' organization.

IV. Approach

A. Background

We first proposed linking software architecture and human collaboration models in our 2012 ICSE New Ideas and Emerging Results track paper [25]. It describes the general idea and approach to achieve co-adaptation and introduces basic concepts. In this paper we focus in detail on the models and mechanism for reflecting runtime software architecture changes in collaboration topologies and how to apply these synchronized views for sophisticated system adaptation.

The co-adaptation of software architecture and human collaboration requires models for specifying the involved runtime elements and their relations. Components and connectors are the primary building blocks of a software architecture. Components are the loci of computation and data management whereas connectors facilitate and control the interactions between components. Based upon Malone and Crowston's observation that human collaboration and software systems share similar coordination requirements [26], we argue for a similar distinction among humans according to work-focused and coordination-focused roles. Along these lines we recently introduced the *human Architecture Description Language* (hADL) for describing collaboration topologies in terms of human components and collaboration connectors [8] (see Fig. 2 and Fig. 3 for examples). Software architecture and human architecture models are thus the core artifacts of our approach.

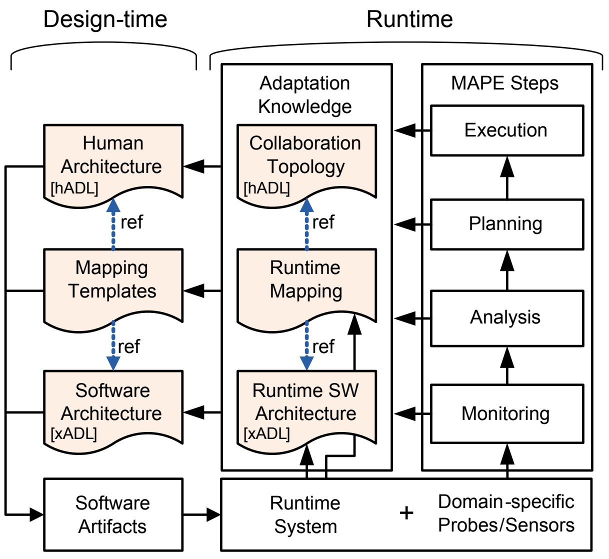

Fig. 4. Reflecting software architecture changes in the human architecture for collaboration-aware system adaptation.

B. Mapping and Adaptation Overview

Given the software architecture and human architecture description, a software architect specifies at design-time how software elements map to collaboration elements and vice-versa. Software architecture-centric events are the primary source for creating a runtime view of the overall system. Our approach aims to leverage these events as much as possible for inferring the collaboration topology (Fig. 4 middle). The mapping specification identifies configurations where software-centric events are insufficient. An event, for example, may describe a new link between an AgentGUI component and a StreamingServer component hosting multiple video streams. While such an event provides sufficient information at the software architecture level, additional information is required to unambiguously connect the respective human agent to a particular SubStream. We thus embed not only software architecture but also human architecture and mapping specification in the software artifacts (Fig. 4 left).

System adaptation typically requires additional domain-specific events besides architecture-centric changes. While independent from the software architecture and collaboration topology, such information describes runtime software and collaboration elements in further detail. Eventually, an adaptation manager utilizes the runtime software architecture, runtime mapping, and collaboration topology in each adaptation step (monitoring, analysis, planning, and execution) to detect and react to critical situations (Fig. 4 right).

C. The Case for an Explicit Architecture Mapping

Multiple generic, extensible, and domain-specific architecture description languages already exist (e.g., ACME [27], xADL [9]) and one could argue that collaboration structures should be embedded at the software architecture level. There are multiple reasons, however, why a separate human architecture model, and thus an explicit, non-trivial mapping, is a better choice:

• Collaboration patterns are sufficiently independent from their implementing software architecture style, and even more so from the detailed software topology. For example, a collaboration system for a rescue task force can be realized as a peer-to-peer system for environments without a communication infrastructure. Alternatively, the client/server style is suitable when a reliable communication infrastructure is available. A collaboration pattern based on supervisors assigning tasks to workers and subsequently collecting their feedback, however, remains in both cases the same. Similarly, the same software architecture style supports different collaboration patterns as demonstrated in the motivating scenario in Section II.

• Software architectures are typically more fine-grained than collaboration structures. Spreading collaboration structure descriptions as annotations across software elements makes it hard to obtain a clear picture of the overall human architecture.

• Structural changes at the collaboration level rarely correspond to structurally equivalent changes at the software level and vice versa. Hence, collaboration changes would remain unnoticed in the software structure, while software topology changes would require additional analysis whether the human architecture remained the same.

• Adaptation relevant properties potentially fit more naturally with hADL elements and thus allow for devising more understandable and manageable adaptation triggers, analysis logic, and adaptation strategies.

• An explicit human architecture keeps the focus on the user and team perspective and thus gives stake-holders an additional model for communicating requirements during the design process. This also enforces a structured approach to explicitly defining adaptation and evolution capabilities at the collaboration level.

V. The Architecture Mapping Process

A. Design-Time Mapping Specification

Synchronizing software architecture and collaboration topology at runtime requires the software architect to specify how software elements map to collaboration elements and vice versa. Our framework utilizes the eXtensible Architecture Description Language (xADL [9]) for describing software component types, connector types, interface types, and containment hierarchies. On the collaboration level, we apply the human Architecture Description Language (hADL [8]) for specifying human component types, collaboration connector types, collaboration object types, collaboration action types, and substructure patterns.

Large-scale systems are typically too dynamic and complex for completely specifying all involved elements and their precise wiring at design-time. Thus, we can neither a-priori fully describe the runtime software structure in xADL nor the collaboration topology in hADL. Consequently, we first define templates that specify for xADL and hADL separately how the various model elements are correctly assembled and connected at runtime. For example, Figure 5 displays on the left a software architecture blueprint for connecting video sources, connectors, streaming server, and video sinks. The dotted frames and lines represent individual templates. The corresponding collaboration topology and templates for the publish/subscribe (human) collaboration pattern are given on the right. We subsequently need additional mapping information to identify which xADL template corresponds to what hADL template (dash-dotted lines in Figure 5).

In other words, a set of xADL, respectively hADL, templates behaves similar to a set of jigsaw pieces: we arrange all pieces according to their shape (i.e., signatures/actions) and obtain a valid overall picture (i.e., architecture). To know which two xADL and hADL pieces go together and where they are supposed to interlock within their respective puzzle, we also need to define matching tabs/blanks (i.e., an *Interlock Point Pair*). See for example the two jigsaw pieces for the Mapping Specification 2 in Fig. 6 left.

Given a software architecture blueprint and collaboration pattern as input, a complete software-to-collaboration mapping specification thus consists of four main parts:

• a set of *xADL elements* (e.g., a StreamingServer component, PublishVideo connector, SubscribeVideo connector, and links from both connectors to the component). The specification refers to the architecture blueprint elements and not the actual element type definition. A type potentially occurs multiple times in a template such as the ReceiveVideoAgent connector and ReceiveVideoWallScreen connector which are both derived from the ReceiveVideo connector type.

• a set of *hADL elements* (e.g., VideoPubSub collaboration connector, SubStream collaboration object, and the link between).

• a set of *Interlock Point Pairs* defines the intersection of two mappings in the software architecture, and where to locate the corresponding interlink at the human architecture level.
A single interlock point pair identifies exactly one xADL interface and exactly one hADL collaboration action. The xADL interface establishes joint points of two xADL puzzle pieces, the hADL action specifies the joint points between two hADL puzzle pieces. Consider the mapping template x1h1 in Fig. 5: the VideoSource's *sendPubStream* interface pairs up with the PubStream's *forward* action.

• the *MappingType* determines how many instances of the

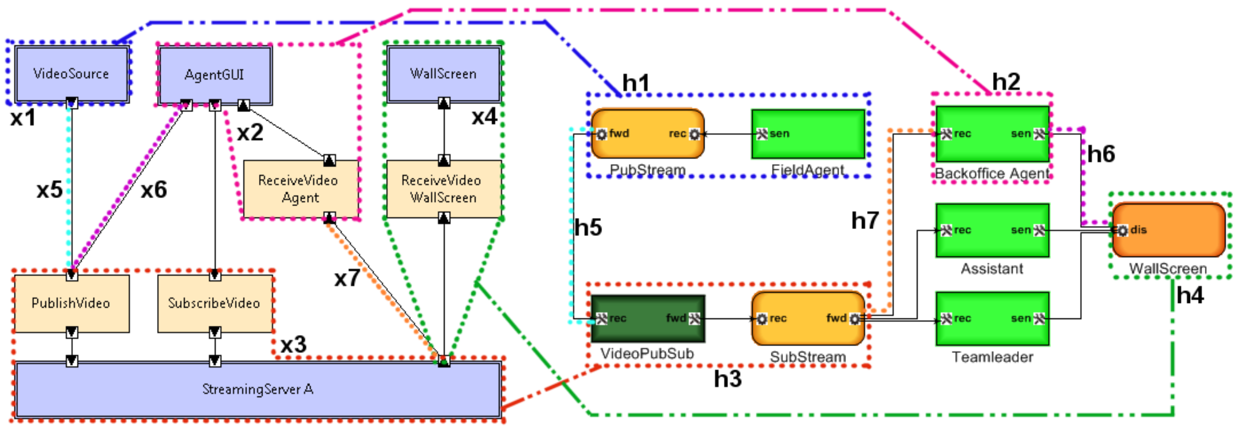

Fig. 5. Example mappings between video streaming architecture blueprint (xADL) and publish/subscribe collaboration pattern (hADL). Mapping 1, for example, consists of xADL element set x1 and hADL element set h1. Mappings for Assistant and Teamleader and corresponding xADL elements are omitted for sake of clarity.

xADL elements may map to how many instances of the hADL elements. In many cases a simple one-to-one mapping will be insufficient. A xADL template instance (e.g., a StreamingServer incl. Connectors) may represent multiple hADL elements (e.g., VideoPubSub collaboration connectors incl. SubStreams). Hence, the following mapping types exist: exact *1-to-1* such as the VideoSource to FieldAgent+PubStream (e.g., x1h2 in Fig. 5), aggregating *1-to-M* (e.g., x3h3 in Fig. 5), replicating *N-to-1* for providing the same video stream on many servers, or a combination thereof (*N-to-M*).

Note that the mapping specification includes only elements that are needed to maintain an unambiguous mapping to collaboration elements. Thus, a software architect typically omits software elements irrelevant to the collaboration topology and vice versa (e.g., the link between the AgentGUI component and the SubscribeVideo connector). For the example in Figure 5, a total of 13 mapping definitions link the software architecture and the collaboration structure (including the six mappings for Assistant and Teamleader not shown).

Our framework leverages software architecture-centric events as much as possible. However, we determine the need for additional *disambiguation events* already at design-time when we derive from the mapping specification that software-level events won't allow for conclusive mapping execution at runtime. At runtime, the link between a VideoSource component and a PublishVideo connector, for example, maps *1-to-M* to the PubStream-to-VideoPubSub link (mapping x5h5 in Fig. 5). Here we need a disambiguation event to define which VideoPubSub (among the many hosted by the StreamingServer) the hADL link should connect to. Applying the jigsaw analogy: non *1-to-1* mappings result in stacked puzzle pieces, for example, mapping instances 1a and 1b in Figure 6. A new mapping instance *4* needs to decide whether linking its hADL piece to mapping *1a* or *1b*. A disambiguation event merely needs to identify any one xADL element involved in the completed hADL piece *4x* and any one hADL element from the targeted, existing hADL piece *1a*.

Identifying at design-time where disambiguation events are required is straightforward: every Interlock PointPair involved in a *1-to-M* or *N-to-M* mapping highlights the need for a corresponding disambiguation event. The developer can then select from the hADL and xADL sets which information will be provide in the disambiguation event. Before system deployment, type information from xADL and hADL models and disambiguation event requirements become embedded in the software artifacts. The exact means (e.g., through source code annotations, middleware configuration, or sensor configuration) remains outside the scope of this paper (see, for example, [28], [3], [6]).

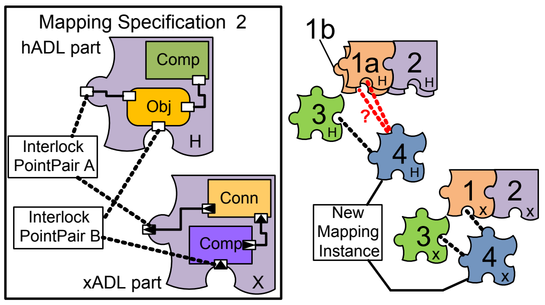

Fig. 6. Utilizing interlock point pair definitions to insert hADL and xADL mappings correctly.

B. Runtime Template Matching and Execution

At runtime, the *Software Architecture Manager* receives system events describing the type and identity of newly deployed software elements, their wiring, respectively their termination, and translates them into software architecture change events (i.e., new/deleted component/connector/link) (Fig. 7 1).

For removal of existing elements, the *Mapping Template Matcher* takes these architecture change events and merely retrieves the respective mapping instance (Fig. 7 2a). For new elements, however, it determines a set of candidate mappings (Fig. 7 2b). Each xADL element type is potentially part of

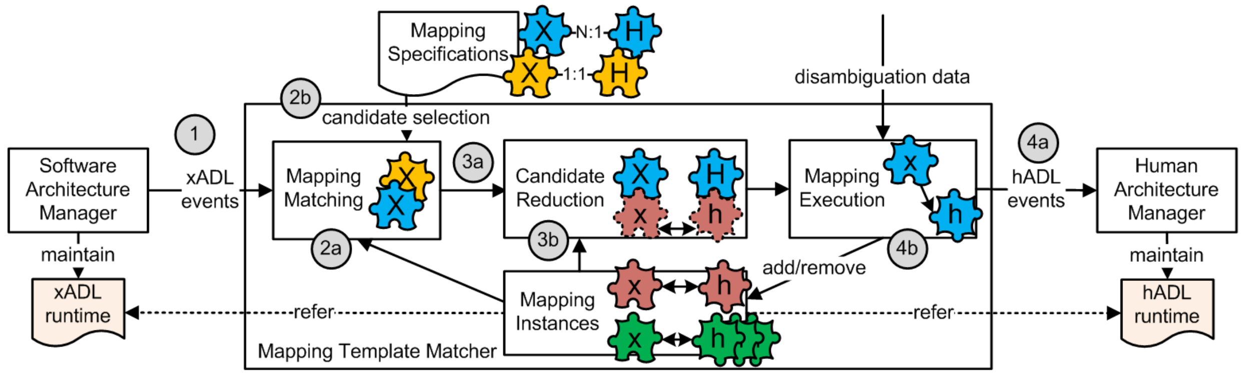

Fig. 7. Artifacts and Steps involved in the Mapping Template Matching process.

multiple mapping definitions (e.g., the link between StreamingServer and ReceiveVideo connectors is of the same type for AgentGUI, AssistantGUI, TeamLeaderGUI, and WallScreen), but ultimately only part of a single mapping instance. The Mapping Template Matcher keeps adding architecture change events to mapping candidates until at least one candidate contains all required xADL elements (Fig. 7 3a). All remaining candidates are discarded. Matching of interlock point pairs with existing neighboring mappings selects the correct mapping in case of multiple simultaneously fulfilled mapping candidates (Fig. 7 3b).

For each completed mapping specification and sufficient disambiguation data, the Mapping Template Matcher dispatches collaboration change events for each mapped hADL element (Fig. 7 4a). When adding new elements, a runtime mapping instance stores references to all involved xADL and hADL instances. *1-to-M* mappings typically accumulate multiple hADL reference sets, respectively *N-to-1* multiple xADL reference sets, and *N-to-M* multiples of both. The Mapping Template Matcher also records interlock point pair instances to track neighboring mapping instances (Fig. 7 4b). Ultimately, the *Human Architecture Manager* processes the collaboration change events to maintain a consistent view of the collaboration topology.

VI. Utilizing hADL for System Self-Adaptation

Correlating software architecture and human architecture offers immense opportunities for sophisticated system sensing, monitoring, analysis, and adaptation (Fig. 8). A system architect utilizes insight into the underlying collaboration pattern at design-time for selecting the appropriate adaptation events, metrics, triggers, and strategies. Later at runtime, the human architecture serves as the data source for exactly those events, metrics, and triggers. Human to software mapping instances subsequently identify the exact software elements requiring reconfiguration.

In this section, we discuss the exemplary application of the runtime human architecture model and its mapping to software architecture elements for system self-adaptation. In line with the motivating scenario, we focus on two exemplary non-functional system requirements and the respective high-level

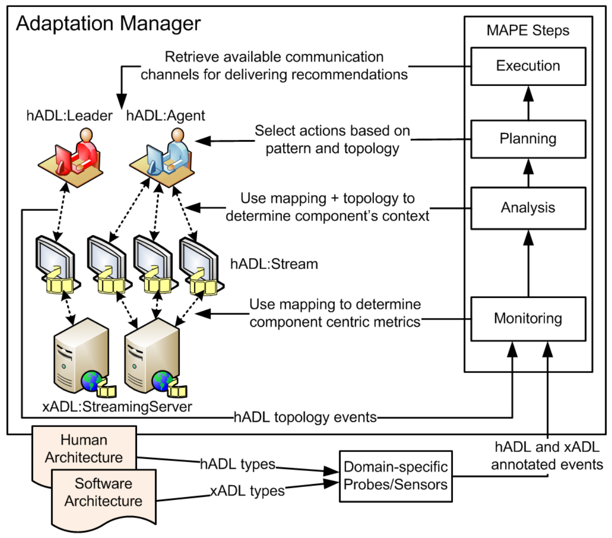
Adaptation Manager
MAPE Steps
Retrieve available communication channels for delivering recommendations
Execution
hADL:Leader
hADL:Agent
Select actions based on pattern and topology
Planning
Use mapping + topology to determine component's context
Analysis
hADL:Stream
Use mapping to determine component centric metrics
Monitoring
xADL:StreamingServer
hADL topology events
Human Architecture
Software Architecture
hADL types
xADL types
Domain-specific Probes/Sensors
hADL and xADL annotated events

Fig. 8. Collaboration-aware system adaptation process.

adaptation approach:
1) Quality: video streams should be available in high resolution: ⇒ limiting the maximum bandwidth usage per StreamingServer component and host.
2) Resilience: system failures should have limited impact on the team's monitoring ability (especially the team leader): ⇒ replicating StreamingServer components and strategically routing streams to avoid single points of failure.

Such adaptation goals require introducing domain-specific data sources. We include xADL hosts that group collocated components and connectors. They also keep track of available bandwidth capacity and consumption. Capturing and processing such additional data remains completely independent of the xADL-to-hADL mapping process.

Sensing
Both architecture views may become carriers of sensor data such as bandwidth constraints. To this end, we extended xADL and hADL with capabilities to store arbitrary system properties. We thus gain the ability to associate sensor data with a particular xADL element, hADL element, or combination thereof.

Video stream bandwidth is an excellent example for a collaboration level property that is relevant for software level adaptation. The software architecture by itself offers no straightforward means for specifying which components, connectors, and links carry a particular video stream. Capturing bandwidth for individual StreamingServer components provides little assistance in determining how to rewire publishers and subscribers to remain within given bandwidth thresholds. Tracking bandwidth usage for individual hADL PubStreams or Substreams, on the other hand, provides promptly the number of consumers, the role of consumers, and via runtime mapping instance data, also the software elements' bandwidth usage.

Tracking a host's bandwidth capacity and utilization complements human architecture-centric bandwidth changes. Hence, changes in (i) video stream bandwidth usage, (ii) a stream's subscription base, and (iii) available host bandwidth may serve as triggers for system adaptation.

Monitoring
Software system monitoring oversees structural and property changes in the software architecture and human architecture. Monitoring can thus enable reassessment of a component's bandwidth usage upon the stream's bandwidth fluctuations as well as changes in the stream's subscriber base.

A system architect applies the mapping specification when creating the monitoring logic to reason how to accurately derive a component's properties. In the case of the StreamingServer's bandwidth usage, the architect aggregates the bandwidth properties of all associated PubStreams and SubStreams multiplied by their subscriber base. Ultimately, monitoring output consists of high-level events and facts such as component bandwidth usage.

Analysis
Software system analysis determines the impact of high-level events such as components exceeding a given bandwidth threshold. Similar to monitoring, system analysis accesses hADL and xADL structures for determining high-level system metrics used later in the planning phase for deciding what adaptation strategies are most suitable. The analysis step ultimately decides whether adaptation is necessary or not.

When a component exceeds its granted bandwidth quota, system analysis collects system properties such as the remaining bandwidth across all hosts. Collaboration-aware algorithms may additionally consider whether the affected component serves streams mainly to the Team leader in a publish/subscribe structure. For the pipes/filter case, an algorithm may determine whether the component becomes a single point of failure when serving streams at the same hop distance from the video source.

For our scenario, system analysis will trigger an adaptation request for a StreamingServer component, supplying information on hosts with sufficient bandwidth capacity, and — depending on the underlying collaboration pattern — determines the component's tendency towards team leader subscriptions or single-point-of-failure, respectively.e.

Planning
The goal of keeping a component's or host's bandwidth usage below a particular threshold applies to the software level and is therefore independent of the underlying collaboration pattern. To this end, the system supports the following fine-grained adaptation actions plans:

1) Replicate the stream at another component and move a (subset of) subscription(s).
2) Move a stream including all subscriptions to another component.
3) Move a subscription to another component already serving the particular stream.
4) Drop a subscription.

The former two plans require a host with sufficient remaining bandwidth (*hostsAvail*), whereas the latter two plans apply when the available bandwidth across all available hosts is exhausted (*hostsFull*).

Given the system analysis' output, planning determines the best adaptation strategy. The particular underlying collaboration pattern constrains how to best perform system reconfiguration while achieving resilience. The runtime collaboration topology determines the applicable set of hosts, components, streams, and subscriptions as input to the adaptation strategies. Due to page constraints we need to limit our discussion of suitable collaboration-aware adaptation strategies to the publish/subscribe pattern.

1) When *hostsAvail*, try separating StreamingService components dedicated to the team leader from regular user components, i.e., a combination of relocating team leader subscriptions to a streaming component serving primarily team leaders and likewise relocating regular subscriptions to components serving regular users.
2) When *hostsFull*, try relocating any team leader subscriptions to an existing stream at another component, otherwise rank streams by their number of subscriptions, and recommend regular subscribers of the most popular streams (i.e., the actual users) to drop their subscription.

The last strategy highlights the potential use of collaboration topologies to include the relevant users in the adaptation of the system when automatic reconfigurations no longer suffice. Again the user selection is collaboration pattern specific: users pull video stream according to a set of properties such as location, quality, or relevance in the publish/subscribe pattern. Hence, recommendations target primarily the stream subscribers to reduce their selection. On the other hand, the pipes/filter pattern has users push video streams to the next consumer. Here recommendations address the stream publishers to be more selective what to forward.

Execution
Enforcing adaptation plans is domain and infrastructure dependent. Research in the domain of autonomic computing and adaptive systems has focused on the execution of software changes for more than a decade (e.g., [6] for architecture-based reconfiguration). On the other hand, autonomous mechanisms and techniques for achieving desired reconfigurations on the collaboration level are limited to a few niche domains (e.g., automatic task management in Amazon Mechanical Turk).

It will require extensive research for evaluating reliability, timeliness, quality, user acceptance, and associated privacy concerns of such adaptation plans. We thus believe that the aspect of actively adapting the human collaboration structure (through autonomic actions, recommendations, or combinations thereof) cannot be sufficiently addressed in the scope of this paper. Nevertheless, we provide a case study in the next section demonstrating the effectiveness approach limited to software system adaptation.

VII. Case Study

In this section we evaluate the added benefit of integrating detailed human architecture knowledge in the adaptation process for a particular scenario. Specifically we are interested in the achievable reliability improvement when applying adaptation strategies tailored to the underlying collaboration topology compared to collaboration-unaware adaptation. In the following, we focus on the publish-subscribe style collaboration topology of the motivating example detailed in Figure 2.

The scenario setup consists of 20 remote high definition video streams at 20 Mbit/s that are randomly connected to nine StreamingServer components; three components per host, three hosts in total (see Fig. 9 for an schematic overview of the software architecture). Agents subsequently subscribe at StreamingService components to receive the desired video streams. The number of subscriptions affects a component's reliability, respectively failure probability. For sake of simplicity we assume a StreamingServer's failure rate p_{fail} to be 0.05 for no subscriptions, linearly increasing to 0.10 when reaching the bandwidth threshold of 150 Mbit/s. Exceeding this threshold triggers adaptation in the form of moving subscriptions (including source streams) among components or spawning new components as long as a host's load remains within a 400 Mbit/s bandwidth limit.

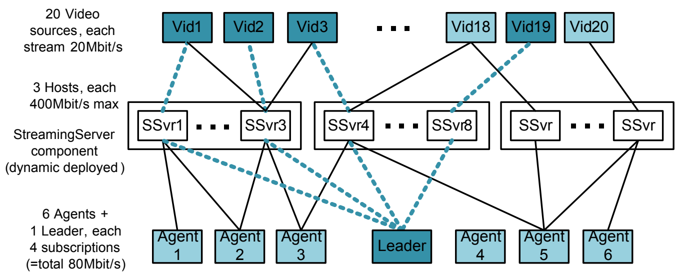
20 Video sources, each stream 20Mbit/s
3 Hosts, each 400Mbit/s max
StreamingServer component (dynamic deployed)
6 Agents + 1 Leader, each 4 subscriptions (=total 80Mbit/s)
Vid1
Vid2
Vid3
Vid18
Vid19
Vid20
SSvr1
SSvr3
SSvr4
SSvr8
SSvr
SSvr
Agent 1
Agent 2
Agent 3
Leader
Agent 4
Agent 5
Agent 6

Fig. 9. Schematic case study configuration (omitting connectors, interfaces, and most links). Dotted lines depict streams to the team leader.

Starting with no subscribers, we gradually increase the number of subscribing agents to six. Each new agent randomly selects two components and chooses two streams each (4 streams total). In-between the regular agent subscriptions, a single team leader connects to a single stream from a randomly selected component for ultimately a total of 4 streams. We assume that sufficient network bandwidth is available.

The adaptation manager observes the component load while the number of subscriptions increases. The collaboration-aware adaptation strategy focuses on moving leader subscriptions to reduce component load as outlined in the previous section. The baseline collaboration-unaware strategy selects subscriptions randomly.

We measure a strategy's impact by determining the average reliability of leader associated streams (dotted lines in Fig. 9). The individual stream reliability $rel(s)$ is determined by the number of stream replicas and the component failure probability of the respective StreamingServer $p_{fail}(comp)$.

$$rel(s) = 1 - \prod_{i=1}^{n} p_{fail}(comp(s_i)) \qquad where\ n = replica(s) \quad (1)$$

Figure 10 compares the achieved average stream reliability for both adaptation strategies as bandwidth usage increases. The chart displays the reliability before and after adaptation for each of the 12 performed reconfigurations (averaging data from multiple experiment runs). The initial spike results from the replication of the first leader subscription. Both adaptation strategies cannot avoid degradation of reliability as the bandwidth load on components and hosts increases. Collaboration-aware adaptation, however, achieves consistently higher reliability through prioritizing leader subscriptions, averaging 0.972 compared to unaware adaptation at 0.952.

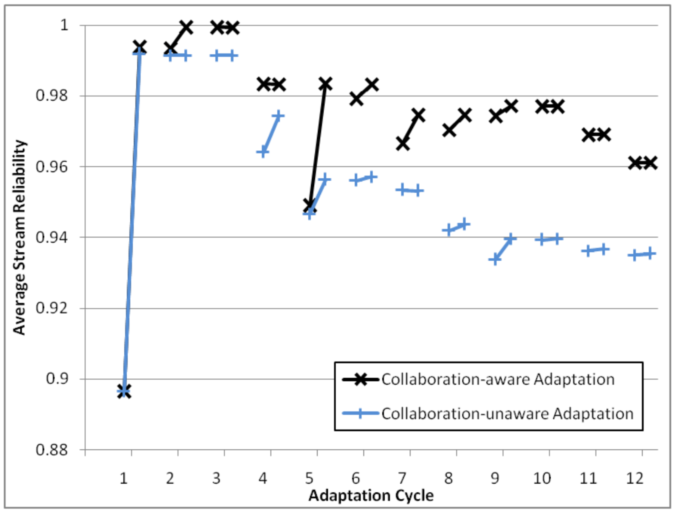

Fig. 10. Average Stream reliability for collaboration-(un)aware adaptation for increasing bandwidth usage.

The adaptation strategies in this case study have been kept simple on purpose, rewiring only the minimum number of subscriptions to bring the bandwidth usage below the threshold. As the results demonstrate, consideration of the human architecture provides significant improvements already for such a simple adaptation approach. We expect algorithms performing even better when taking advantage of the full extent of the collaboration topology.

VIII. Conclusions

We presented our approach for linking software architecture and collaboration topology for enabling more sophisticated system adaptation. System adaptation remains unaware of collaboration interdependencies without such mapping information. To this end, we provided a software architecture to human architecture mapping specification at design-time and a framework for reflecting software architecture events in the human architecture at runtime. We further make the case for integrating the collaboration topology at all stages of the MAPE-K adaptation cycle. Our case study demonstrates the benefit of our approach.

Next steps will focus on supporting the mapping process in detecting inconsistencies and incomplete coverage as the prerequisite to providing adaptation guarantees. While our current work focused primarily on adapting the software system, future research will address the challenge of adapting also the human architecture. We will investigate how autonomic adaptation actions and recommendations can be combined for achieving desirable system configurations. Simultaneously, we propose applying the MAPE-K process on the human architecture for addressing undesirable human collaboration situations. A collaboration-centric adaptation mechanism may observe, for example, how many users access the wall screen and recommend a suitable access coordination mechanism.

Acknowledgment

This work is supported in part by the Austrian Science Fund (FWF) under Grant No. J3068-N23 and the National Science Foundation under Grant No. 0917129.

References

[1] L. Northrop, P. Feiler, R. P. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, D. Schmidt, K. Sullivan, and K. Wallnau, "Ultra-Large-Scale Systems - The Software Challenge of the Future," Software Engineering Institute, Carnegie Mellon, Tech. Rep., June 2006.

[2] J. Kramer and J. Magee, "Self-managed systems: an architectural challenge," in *International Conference on Software Engineering*, 2007, pp. 259–268.

[3] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf, "An architecture-based approach to self-adaptive software," *IEEE Intelligent Systems*, vol. 14, pp. 54–62, May 1999.

[4] B. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee *et al.*, "Software engineering for self-adaptive systems: A research roadmap," in *Software Engineering for Self-Adaptive Systems*, ser. Lecture Notes in Computer Science, B. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Springer, 2009, vol. 5525, pp. 1–26

[5] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven, "Using architecture models for runtime adaptability," *Software, IEEE*, vol. 23, no. 2, pp. 62 – 70, march-april 2006.

[6] J. C. Georgas and R. N. Taylor, "Policy-based architectural adaptation management: Robotics domain case studies," in *Software Engineering for Self-Adaptive Systems*, B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Springer, 2009, pp. 89–108.

[7] D. A. Menasce, J. a. P. Sousa, S. Malek, and H. Gomaa, "Qos architectural patterns for self-architecting software systems," in *Proceedings of the 7th international conference on Autonomic computing*, ser. ICAC '10. New York, NY, USA: ACM, 2010, pp. 195–204.

[8] C. Dorn and R. N. Taylor, "Architecture-driven modeling of adaptive collaboration structures in large-scale social web applications," in *Int. Conf. on Web Information Systems Engineering (WISE)*, Nov. 2012.

[9] E. M. Dashofy, A. v. d. Hoek, and R. N. Taylor, "A comprehensive approach for the development of modular software architecture description languages," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, pp. 199–245, April 2005.

[10] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, no. 10, pp. 46 – 54, oct. 2004.

[11] J. Dowling and V. Cahill, "The k-component architecture meta-model for self-adaptive software," in *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*. London, UK: Springer-Verlag, 2001, pp. 81–88.

[12] J. Zhang and B. H. C. Cheng, "Model-based development of dynamically adaptive software," in *ICSE '06: Proceedings of the 28th international conference on Software engineering*. New York, NY, USA: ACM, 2006, pp. 371–380.

[13] W. Cazzola, A. Ghoneim, and G. Saake, "Software evolution through dynamic adaptation of its oo design," in *Objects, Agents and Features: Structuring Mechanisms for Contemporary Software, Lecture Notes in Computer Science*. Springer-Verlag, 2004, pp. 69–84.

[14] D. Sykes, W. Heaven, J. Magee, and J. Kramer, "From goals to components: a combined approach to self-management," in *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, ser. SEAMS '08. New York, NY, USA: ACM, 2008, pp. 1–8.

[15] G. Edwards and N. Medvidovic, "A methodology and framework for creating domain-specific development infrastructures," in *Automated Software Engineering, International Conference on*. Los Alamitos, CA, USA: IEEE Computer Society, September 2008, pp. 168–177.

[16] D. Di Ruscio, I. Malavolta, H. Muccini, P. Pelliccione, and A. Pierantonio, "Developing next generation adls through mde techniques," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 85–94.

[17] N. Esfahani, S. Malek, J. a. P. Sousa, H. Gomaa, and D. A. Menascé, "A modeling language for activity-oriented composition of service-oriented software systems," in *Int. Conf. on Model Driven Engineering Languages and Systems*, MODELS '09. Springer, 2009, pp. 591–605.

[18] A. Bhave, D. Garlan, B. Krogh, A. Rajhans, and B. Schmerl, "Augmenting software architectures with physical components," in *Proceedings of the Embedded Real Time Software and Systems Conference (ERTS2 2010)*, 2010.

[19] P. Wohed, W. van der Aalst, M. Dumas, A. ter Hofstede, and N. Russell, "On the suitability of bpmn for business process modelling," in *Business Process Management*, Lecture Notes in Computer Science, Springer, 2006, vol. 4102, pp. 161–176.

[20] M. Ford, A. Endpoints, and C. Keller, "Ws-bpel extension for people (bpel4people), version 1.0," 2007.

[21] C. Dorn and R. N. Taylor, "Analyzing runtime adaptability of collaboration patterns," in *International Conference on Collaboration Technologies and Systems (CTS)*. CO, USA: IEEE Computer Society, 2012.

[22] D. Schall, H.-L. Truong, and S. Dustdar, "Unifying human and software services in web-scale collaborations," *IEEE Internet Computing*, vol. 12, no. 3, pp. 62–68, 2008.

[23] M. Conway, "How do committees invent," *Datamation*, vol. 14, no. 4, pp. 28–31, 1968.

[24] M. Cataldo, P. A. Wagstrom, J. D. Herbsleb, and K. M. Carley, "Identification of coordination requirements: implications for the design of collaboration and awareness tools," in *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, ser. CSCW '06. New York, NY, USA: ACM, 2006, pp. 353–362.

[25] C. Dorn and R. N. Taylor, "Co-adapting human collaborations and software architectures," in *Proceedings of the 34th international conference on Software engineering (ICSE)*, 2012, pp. 1277–1280.

[26] T. W. Malone and K. Crowston, "The interdisciplinary study of coordination," *ACM Comput. Surv.*, vol. 26, pp. 87–119, March 1994.

[27] D. Garlan, R. Monroe, and D. Wile, "Acme: an architecture description interchange language," in *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, ser. CASCON '97. IBM Press, 1997, pp. 7–.

[28] P. Oreizy, N. Medvidovic, and R. N. Taylor, "Architecture-based runtime software evolution," in *International Conference on Software engineering*, ICSE '98. Washington, DC, USA: IEEE, 1998, pp. 177–186.