A Domain-Specific Language for Coordinating Collaboration

Christoph Mayr-Dorn
Institute for Software Systems Engineering
Johannes Kepler University, Linz, Austria
Email: christoph.mayr-dorn@jku.at

Christoph Laaber
Department of Informatics
University of Zurich, Switzerland
Email: laaber@ifi.uzh.ch

Abstract

Manually managing collaboration becomes a problem in distributed software engineering environments. Individual engineers easily loose track of who to involve and when. The result is lack of communication, alternatively communication overload, leading to errors and rework. This paper presents a Domain-Specific Language (DSL) for scripting of collaboration structures and their evolution. We demonstrate the DSL's benefits and expressiveness for setting up an iteration planning meeting in an agile development setting.

1. Introduction

Development of non-trivial software is typically a collaborative effort — regardless of whether a rigid waterfall process or agile methods are followed. A plethora of engineering support tools focus on collaboration around joint tasks or software artifacts [1] - most of them requiring the engineers to take initiative in managing the collaboration.

Manually managing collaboration becomes a problem in complex engineering setting, especially when teams are distributed [2]. Take a distributed agile team as an example where customer, business analysts, and developers are no longer colocated but need to collaborate on vital activities such as an iteration planning meeting (IPM). Individual engineers loose track of who to involve [3], for what purpose, and for what duration. The result is lack of communication, respectively, communication overload, leading to errors and rework.

We propose a Domain-Specific Language (DSL) for coordinating collaboration. Our DSL enables scripting of collaboration structures and their evolution: specifying for individual tasks which collaboration mechanisms to use, who to involve, and what collaboration-specific capabilities the participants have. Example collaboration mechanisms employed in Software Engineering (SWE) are wikis, chat rooms, collaborative document editors, issue tracking, or source code version control systems. Previous work focuses on revealing collaboration post-hoc, respectively focuses on managing tasks. Our approach aims for a complementary view for coordinating collaboration alongside any process-centric engineering methodology.

2. Scripting Collaborations

We propose to support collaboration in SWE environments through a DSL that simplifies setting up collaboration structures. The DSL has the primary purpose to:

- address collaboration participants (e.g., identify and contact engineers to participate in an IPM),
- configure collaboration mechanisms (e.g., setting up a chat or wiki page),
- navigate across the relations among process artifacts (e.g., identify test cases that link to a user story on a wiki page),
- manipulate the relations among collaboration mechanisms (e.g., link a chat room to a user story),
- as well as manipulate the participants' collaboration capabilities (e.g., make the Scrum master the moderator of a chat room).

Creating a DSL that is able to integrate arbitrary collaboration mechanisms is challenging. The various collaboration mechanisms come with different collaboration semantics — consider the difference between chatting, artifact editing, and issue assigning. In addition, collaboration tools take different conceptual approaches to providing an API. Our DSL supports the scripting effort by abstracting from low-level, technical tool APIs and high-level collaboration semantics through a common representation (i.e., hADL, the human Architecture Description Language [4]). With hADL, we specify the building blocks (e.g., the roles, the collaboration mechanisms, what collaboration capabilities these mechanisms offer to individual roles, ...) from which to build a collaboration structure. Our scripting environment then uses hADL models to ensure correct construction of collaboration structures while remaining completely independent from any particular hADL model instance.

Figure 1 depicts the high-level approach for using our DSL. Our approach considers three roles. The *Script Author* loads a hADL model within the DSL *Scripting Environment*. Any stakeholder in the engineering process may assume the role of script author, for example the Scrum master. While scripting, the environment continuously checks the script for compliance with the hADL model, assists through model-centric script completion, and generates for correct script segments the corresponding Java code. At runtime, the (potentially automated) *Script Activator* role triggers this Java code, which in turn calls the *hADL Runtime Framework* [5]. The execution framework in turn interacts with the actual collaboration tools and contacts the *Collaboration Participants*. A stakeholder might assume multiple roles at the same time. For example, the Scrum master

978-1-5386-2141-7/17 $31.00 © 2017 IEEE
DOI 10.1109/SEAA.2017.33

might author the IPM script, activate the script prior to the meeting, and ultimately become involved as a chat participant.

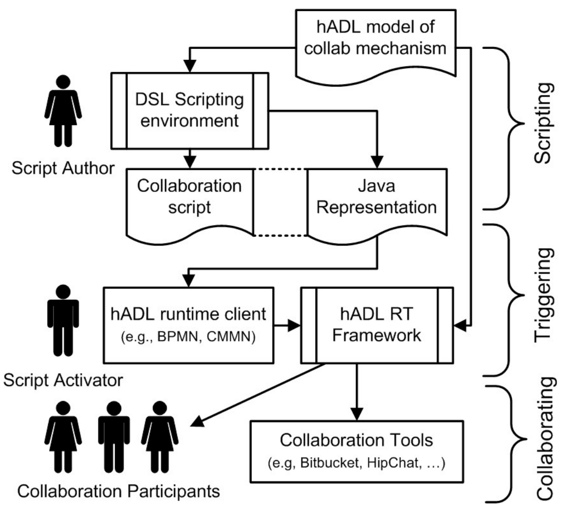
hADL model of collab mechanism
DSL Scripting environment
Script Author
Collaboration script
Java Representation
Scripting
hADL runtime client (e.g., BPMN, CMMN)
hADL RT Framework
Triggering
Script Activator
Collaboration Tools (e.g, Bitbucket, HipChat, ...)
Collaboration Participants
Collaborating

Fig. 1. Approach for scripting of collaboration structures.

2.1. DSL Elements

We introduce the DSL elements based on script excerpts. Listings print DSL keywords in **colored, bold font** and hADL model instance types in *italics*.

2.1.1. Tasks. The script file is the implicit top-level DSL element. It contains a collection of `Tasks`. A task represents a logical step for coordinating collaboration and to this end composes multiple fine-grained, interdependent collaboration structure changes. A task's complexity ranges from simply setting up a chat room to preparing a complete IPM.

Listing 1 exemplifies the general structure of a task definition. Tasks have a name (line1), an optional set of input parameter declarations (line 2-4), a body of statements (line 5-7), and, finally, a set of optional output parameter declarations (line 8). A collaboration script typically consists of one or more tasks, e.g., one each for creating a chatroom, adding chatroom participants, and tearing down the chatroom.

2.1.2. Input, Output, and Local Variables. Input parameters specify what information a task requires. The DSL distinguishes between primitive Java type parameters and hADL type parameters to support hADL model-backed validity checks and code completion. The former allow the script activator to customize hADL elements, for example, passing a string to be used as the topic of a chatroom. The latter enables passing in hADL element instances that were created in a preceding script task. For example, for adding participants to a preexisting chatroom, the script activator passes the chatroom hADL instance together with the participant hADL instances. The hADL model element's id (e.g., *scrum.obj.ChatRoom*) serves as type information. Square brackets around parameters indicate lists.

A local variable (keyword `var`) stores intermediary results for further processing, navigation, iteration, and ultimately output. Output parameters specify what a task will pass back as a result of successful execution. The DSL supports hADL type output parameters only. In the example Listing 1 line 6, the variable `links` collects the CollaborationLinks (of type *scrum.links.Chatting*) established when adding new participants to the chat room.

```
1  task change_chatroom {
2    in nUsers : [scrum.roles.ChatUser]
3    in room : scrum.obj.ChatRoom
4    javaIn topic : String
5    ...
6    var links : [scrum.links.Chatting]
7    ...
8    out links as "membership" }
```

Listing 1. Example Variable, Input, and Output

2.1.3. Manipulation Primitives. The DSL provides manipulation primitives (see Listing 2) for setting up, changing, and tearing down a collaboration structure. The `acquire` keyword obtains a new hADL element instance of given type using a resource descriptor (line 1). In our example, the resource descriptor *rd* contains the chatroom's topic and privacy settings. Eventually, the script author uses the `release` keyword to remove a hADL instance, here tearing down the chatroom (line 2).

Adding an engineer to a chatroom or making an engineer the editor of a wikipage occurs via the `link` keyword (line 4). The script editor specifies the human component instance, the collaboration object to link to, and the collaboration link type. The DSL automatically restricts the link type to valid ones. For example, given that *alice* is of type *scrum.roles.ChatUser*, only the link type *scrum.links.Chatting* would be selectable. The `unlink` keyword removes the link, here effectively removing alice's chatroom membership (line 5).

Establishing and removing collaboration references happens in the same manner as manipulating collaboration links but involves two collaboration object instances. Here, line 7 establishes a `reference` between a task and a wikipage, subsequently removed on line 8.

```
1  acquire scrum.obj.ChatRoom with rd as chatRoom1
2  release chatRoom1
3  ...
4  link alice and chatRoom1 by scrum.links.Chatting as
       chatLink1
5  unlink chatLink1
6  ...
7  reference from task1 to wikiPage2 with
       scrum.rel.HasOutput as ref1
8  dereference ref1
```

Listing 2. Example use of Manipulation Primitives

2.1.4. Navigation and Iteration. Navigating a collaboration structure (see Listing 3) allows the script author to select connected hADL elements without having to address each one with a dedicated `acquire` operation and explicit resource descriptor. Line 1 exemplifies how to retrieve all tasks contained in a particular sprint. Line 2 demonstrates how to chain multiple references to obtain all wikipages related to a sprint. Our DSL supports navigation across collaboration links and references.

The DSL scripting environment supports the script author by providing a choice of valid collaboration links and references. This reduces the mental effort of the script author when defining navigation paths across intermediate object types. After navigating to and loading a set of hADL element instances, the script author may iterate over these elements in a `for all` loop. All DSL statements are available within the loop's body. Listing 3 (lines 4-10) demonstrates how to obtain all editors of a wikipage and adding them to a chat room.

```
1  startingFrom sprint1 load scrum.obj.ScrumTask:
       scrum.rel.ContainsTask as tasks
2  startingFrom sprint1 via scrum.obj.ScrumTask:
       scrum.rel.ContainsTask load scrum.obj.WikiPage:
       scrum.rel.HasOutput as pages
3
4  var chatMemberships : [scrum.links.Chatting]
5  startingFrom page load scrum.roles.ScrumUser:
       scrum.links.EditingPage as editors
6  for all editors as editor {
7    acquire scrum.roles.ChatUser with editor.RD as
       chatUser1
8    link chatUser and chatRoom1 by scrum.links.Chatting
       as chatLink1
9    add chatLink1 to chatMemberships }
```

Listing 3. Example use of Navigation and Iteration

2.1.5. DSL Tool Support and Script Integration. We implemented the DSL with the Eclipse-based XText framework. It provides a basic DSL parser, linker, compiler, and type-checking infrastructure. Aside from specifying the DSL grammer, our main effort focused on providing model-centric DSL code auto-completion, syntax proposals, constraint validation, and Java code generation.

The resulting scripting environment produces a Java class for each script, where each DSL `Task` becomes a method accepting the specified input parameters. To accommodate multiple task output parameters, the Java method returns output as parameter/value pairs in a map. Triggering a task becomes simply invoking the corresponding method.

3. Case Study

Our case study demonstrates the DSL's expressiveness for setting up complex collaboration structures. Specifically, we show a detailed example script for realizing part of an exemplary iteration planning meeting. Second, the case study demonstrates that it is technically possible to script (and with the help of the hADL runtime framework also execute) collaboration setup, manipulation, and tear down actions. DSL and scripting environment including validity checks, models, as well as hADL runtime plugins are outlined in further detailed at goo.gl/Ex4GHu.

We support the core Scrum elements through (reusable) plugins for the hADL runtime framework for Agilefant, Bitbucket, HipChat and Google Drive Documents.

For our use case, we assume that the spatially distributed agile team members utilize a dedicated *ScrumTask* as container for all meeting preparations. Analysts (in the role of *ScrumUser*) collect test scripts in *WikiPages*. These wiki pages link to the documents where domain experts (in the role of *DocUsers*) sketch out the stories.

The preparation *ScrumTask* serves as entry point for the script in Listing 4 (line 2,5). Additional required input is the topic and the wiki page name for the IPM's chatroom, respectively meeting minutes (line 3,4,7,8).

From the entry task, the script retrieves the owning masters (line 8) and adds them as owners of the IPM's minutes (line 9-10). Also from the entry task, the script navigates to all linked wiki pages, i.e, containing test scripts (line 11), to retrieve their editors (line 13) and adds them as chat participants (line 14-16). From the wiki pages, the script also navigates to the associated documents, i.e. containing the story sketches, and similarly adds the document owners as chat participants (line 17-20). Ultimately, the script returns the created chat room and list of organizers (line 21-22).

```
1  task setupIPM {
2    javaIn ipmPrepTaskId : Integer
3    javaIn chatTopic : String
4    javaIn minutesPage : String
5    acquire scrum.obj.ScrumTask with
       RDFactory.agilefant(ipmPrepTaskId) as
       ipmPrepTask
6    acquire scrum.obj.WikiPage with RDFactory.bitbucket
       (minutesPage) as minutes
7    acquire scrum.obj.ChatRoom with RDFactory.hipChat(
       chatTopic) as ipmChat
8    startingFrom ipmPrepTask load
       scrum.roles.ScrumMaster:
       scrum.links.MasterEditingTask as masters
9    for all masters as master {
10     link master and minutes by scrum.links.OwningPage
        as owningPageLink }
11   startingFrom ipmPreptask load scrum.obj.WikiPage:
       scrum.rel.HasOutput as pages
12   for all pages as page {
13     startingFrom page load scrum.roles.ScrumUser:
       scrum.links.EditingPage as editors
14     for all editors as editor {
15       acquire scrum.roles.ChatUser with editor.RD as
       chatUser1
16       link chatUser1 and ipmChat by
       scrum.links.Chatting as chatLink }
17     startingFrom page via scrum.obj.WikiPage:
       scrum.rel.DependsOn load scrum.roles.DocUser:
       scrum.links.OwningDoc as docOwners
18     for all docOwners as owner {
19       acquire scrum.roles.ChatUser with owner.RD as
       chatUser2
20       link chatUser2 and ipmChat by
       scrum.links.Chatting as chatLink }}
21   out ipmChat as "ipmChatRoom"
22   out masters as "organizers" }
```

Listing 4. IPM Setup Script

The DSL provides three main benefits. First, the script author can focus on defining the collaboration structure without having to know tool-specific implementation details. S/he requires only minimal, high-level collaboration tool know-how (i.e., how to correctly create the resource descriptors) to devise complex collaboration structures. The abstraction from the underlying collaboration tools provides the opportunity to replace these tools without having to rewrite the scripts. A company could thus more easily switch from, for example, HipChat to Slack.

Second, scripts capture collaboration best practices (e.g., how to identify and involve relevant people and artifacts for an IPM meeting) and thus are ideal to capture otherwise tacit knowledge. We acknowledge that learning the DSL and writing DSL scripts requires a non-negligible effort. This effort pays off quickly when several teams within a larger software development organization share and reuse these scripts. In a large company such as Google with thousands of teams the DSL, thus, provides a means to spread and harmonize this tacit collaboration know-how.

Third, the DSL provides suggestions what collaboration manipulation actions are available from a given collaboration element respectively ensures that only valid elements are selected. In the example script in Listing 4 the DSL conducts 24 validity checks, a significant amount of otherwise mental checks even in such a short script.

4. Related Work

Over the last two decades, the software engineering community repeatedly investigated the need for combining process technology and collaboration support [6], [7]. Supporting communication and coordination in distributed engineering settings, however, remains a serious challenge [8].

In recent work, Zhao et al. apply Little-JIL for describing fine-grained steps involved in refactoring [9]. While this approach captures which artifacts were changed by what rework activity, the involved participants and collaboration dependencies remain implicit. Kedji et al. [10] provide a collaboration-centric development process model and corresponding DSL. Their DSL focuses on roles and work assignments using artifacts; however no other collaboration mechanism is supported. A major additional difference is, we don't aim to establish authoritative data but restrict ourselves to integrating existing tools. The GENESIS environment [11] integrates collaboration tools but doesn't allow control over the collaboration mechanisms themselves. Esfahani et al. [12] introduce agile method fragments which include collaborative activities such as pair programming, daily meeting, or inspection. These fragments are not executable and don't specify the topology of the corresponding collaboration structures. In general, collaborative development environments often integrate, among other tools, issue management, tasks and collaboration mechanisms [13].

In summary, existing approaches are unable to support collaboration beyond accessing engineering artifacts within an integrated engineering environment. To the best of our knowledge, our DSL is the first attempt at scripting of collaboration structures in software engineering settings.

5. Conclusions

We made the case for supporting the scripting of collaboration structures in software engineering environments. Even highly agile settings such as Scrum processes benefit from prepared scripts. These reduce the risk of, for example, overlooking relevant collaboration participants or artifacts as may happen in purely ad-hoc, human-driven collaborations. Our DSL and scripting environment utilize hADL models for assisting the script author in correctly creating, modifying, and tearing down complex collaboration structures.

The current DSL contains the basic features for collaboration scripting. Further evaluation in the field is required to understand what advanced features software engineers might find useful. Future work also includes exploration how to automatically integrate the generated Java code with development-centric tools.

Acknowledgments

This work was supported by the Austrian Science Fund (FWF): P29415-NBL funded by the Government of Upper Austria.

References

[1] J. Portillo-Rodríguez, A. Vizcaíno, M. Piattini, and S. Beecham, "Tools used in global software engineering: A systematic mapping review," *Information and Software Technology*, vol. 54, no. 7, pp. 663–685, 2012.

[2] I. Mistrík, J. Grundy, A. Van der Hoek, and J. Whitehead, "Collaborative software engineering: Challenges and prospects," in *Collaborative Software Engineering*. Springer, 2010, pp. 389–403.

[3] A. Mockus and J. D. Herbsleb, "Expertise browser: a quantitative approach to identifying expertise," in *Proc. of the 24th Int. Conf. on Software Engineering*. ACM, 2002, pp. 503–512.

[4] C. Dorn and R. N. Taylor, "Architecture-Driven Modeling of Adaptive Collaboration Structures in Large-Scale Social Web Applications," in *WISE*, LNCS vol. 7651. Springer, 2012, pp. 143–156.

[5] C. Mayr-Dorn and S. Dustdar, "A framework for model-driven execution of collaboration structures," in Proc. of *Int. Conf. on Advanced Information Systems Engineering , CAiSE 2016*, LNCS vol. 9694. Springer, 2016, pp. 18–32.

[6] P. Barthelmess, "Collaboration and coordination in process-centered software development environments: a review of the literature," *Inf. and Softw. Techn.*, vol. 45, no. 13, pp. 911 – 928, 2003.

[7] V. Ambriola, R. Conradi, and A. Fuggetta, "Assessing process-centered software engineering environments," *ACM Trans. on Software Engineering and Methodology (TOSEM)*, vol. 6, no. 3, pp. 283–328, 1997.

[8] B. Sengupta, S. Chandra, and V. Sinha, "A research agenda for distributed software development," in *Proc of Int. Conf on Software engineering*. ACM, 2006, pp. 731–740.

[9] X. Zhao and L. Osterweil, "An approach to modeling and supporting the rework process in refactoring," in *ICSSP*, june 2012, pp. 110 –119.

[10] K. A. Kedji, R. Lbath, B. Coulette, M. Nassar, L. Baresse, and F. Racaru, "Supporting collaborative development using process models: An integration-focused approach," in *ICSSP*, 2012, pp. 120–129.

[11] L. Aversano, A. De Lucia, M. Gaeta, P. Ritrovato, S. Stefanucci, and M. Luisa Villani, "Managing coordination and cooperation in distributed software processes: the genesis environment," *Software Process: Improvement and Practice*, vol. 9, no. 4, pp. 239–263, 2004.

[12] H. C. Esfahani and E. Yu, "A repository of agile method fragments," in *Int. Conf. on Software Process*. Springer, 2010, pp. 163–174.

[13] F. Lanubile, C. Ebert, R. Prikladnicki, and A. Vizcaíno, "Collaboration tools for global software engineering," *IEEE software*, vol. 27, no. 2, p. 52, 2010.