Statelets: Coordination of Social Collaboration Processes

Vitaliy Liptchinsky, Roman Khazankin, Hong-Linh Truong, and Schahram Dustdar

Distributed Systems Group, Vienna University of Technology
Argentinierstrasse 8/184-1, A-1040 Vienna, Austria
{lastname}@infosys.tuwien.ac.at

Abstract. Today people work together across time, space, cultural and organizational boundaries. To simplify and automate the work, collaboration employs a broad range of tools, such as project management software, groupware, social networking services, or wikis. For a collaboration to be effective, the actions of collaborators need to be properly coordinated, which requires taking into account social, structural, and semantic relations among actors and processes involved. This information is not usually available from a single source, but is spread across collaboration systems and tools. Providing a unified access to this data allows not only to establish a complete picture of the collaboration environment, but also to automate the coordination decision making by specifying formal rules that reflect social and semantic context effects on the ongoing collaboration processes. In this paper we present Statelets, a coordination framework and language for support and coordination of collaboration processes spanning multiple groupware tools and social networking sites, and demonstrate its suitability in several use cases.

Keywords: Coordination Language, Collaboration, Social Context, Groupware Integration.

1 Introduction

Groupware and social software foster collaboration of individuals who work across time, space, cultural and organizational boundaries, i.e., virtual teams [22]. Problem of people coordination in collaborative processes has been already extensively studied in academia, (e.g., in [7,9]), and addressed in industry with ever more groupware products incorporating workflow and orchestration mechanisms (e.g., Microsoft Sharepoint). However, in many cases, people interact and contribute in divergent commercial or non-profit on-line collaboration platforms, such as social networks, open source development platforms, or discussion forums, that remain decoupled, isolated and specific to their domains. The problem of coordination in such a setup gets a new look, where processes that need to be coordinated are decentralized and distributed across different specialized tools and online services.

M. Sirjani (Ed.): COORDINATION 2012, LNCS 7274, pp. 1–16, 2012.

Social network context is an integral part of human coordination. For example, the following context aspects have an impact on the behavior of collaborating individuals: actions taken by neighbors in social network [10], social neighbors' preferences [4], and the social network structure itself [25]. The degree of the impact varies from network context simply 'carrying' the information that can be used in a process to forcing adjustment or even cancellation of ongoing actions. Also, social context can imply mutual dependency between processes, reflected by such common coordination mechanisms in social networks as collective actions [4], i.e., 'I'll go if you go'. Social network context can be used for such advanced activities as expertise location [17], composition of socially coherent collaborative teams [5], discovery of unbiased reviewers, and so on.

Along with the social component of the network context, semantic relations between processes may affect coordination decisions as well. In groupware and wiki-like platforms, processes are reflected as incremental changes of common deliverables (e.g., documentation of an idea, a technical specification, or a source code file) connected into dependency and semantic networks. Relations between these artifacts may influence the collaboration process. For example, actions on a document should not be performed before related documents reach a certain condition, or a change in a related document might force to re-do an activity.

Due to an information-centric nature of both social and semantic contexts, we combine these notions together and define *network context of a collaboration process* as *information about related processes and people, their actions and states.* In our previous work [15] we discussed network context effects on collaboration processes, and presented an approach for modeling them.

In spite of growing interest to social network effects in academia [4,10,25], the problem of network context-based coordination has not been properly addressed by coordination languages and frameworks. As examined in the paper, existing coordination languages lack necessary features to enable efficient programming of coordination based on network effects. We refer here to suitability as an amount of efforts a developer needs to spend to express such coordination rules. Also, supplying the developer with social and semantic network context requires horizontal composition of groupware and social networking sites, which imposes yet additional challenges [6,14], which are not addressed properly by existing frameworks as well.

In this paper we present Statelets, a programming language for coordination of social collaboration processes spanning multiple software systems. A distinguishing characteristic of Statelets is the support for coordination based on social and semantic network effects. Although the primary focus of the paper is the programming language, our contribution also includes a conceptual architecture of the underlying framework that aims at integration of groupware and social networks to extract social and semantic contexts. To evaluate Statelets, we have implemented use cases that show its advantages and suitability to the domain.

The rest of this paper is structured as follows: Section 2 provides a motivating example and identifies the features that are crucial for a network context-based coordination language. In Section 3 we explore the suitability of existing coordination

languages for the problem at hand in the perspective of these features. Sections 4 and 5 describe the Statelets coordination language and the conceptual architecture of the underlying framework respectively. Section 6 demonstrates the usage of the language with use cases. The paper is concluded in Section 7.

2 Motivation

As a motivating example, let us consider open source software engineering. Projects in software engineering can be classified into analysis projects and engineering projects (See Fig. 1b). An analysis project represents a non-routine and changeable process, whereas an engineering project represents a rather routine and stable process. Both types of projects produce deliverables, such as source code or technical documentation. Projects get assigned to members of open source communities, who are located via social (professional) networks and online collaboration services, and are then hold responsible for the progress of corresponding activities.

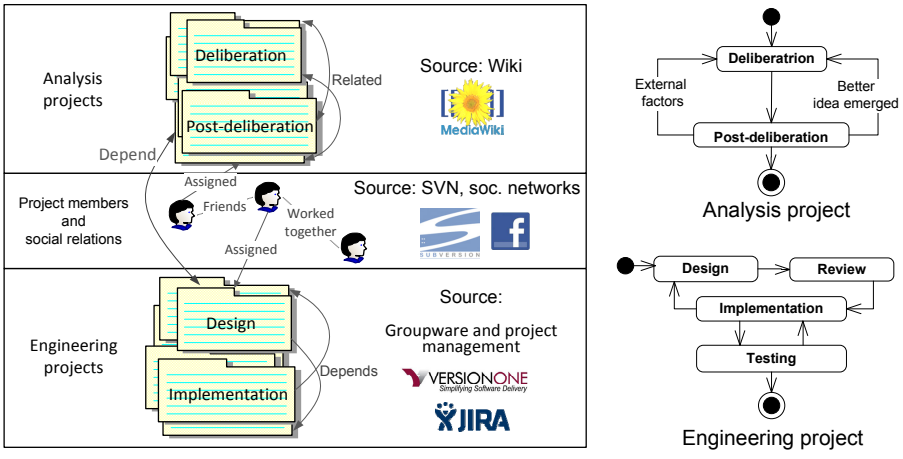
Projects can be related to or depend on each other. For example, two projects are related if they contribute to the same software product, are functionally interdependent, or share components, goals, or resources. Similarly, social and professional relations and technical dependencies exist between project members, e.g., a software engineer depends on engineers who wrote previous versions of the component or worked on the code in the past. Figure 1a depicts various relations between projects and their members.

The key to success of such engineering and analysis projects are advanced activities, such as expertise and resource discovery. Such activities are not possible without integration of professional (e.g., XING, LinkedIn) and private (e.g., Facebook, MySpace) social networks, and online collaboration tools (e.g., SourceForge). Figure 1a depicts integration and execution environment of processes that correspond to analysis and engineering projects. Engineering projects are more specific to the domain, and, therefore, require more specific groupware, e.g., VersionOne, or Jira. Analysis projects, on the contrary, require more flexible and wide-spread groupware, such as MediaWiki (engine for Wikipedia).

Given the setup described above, let us consider the following possible coordination rules:

1. *If an Analysis project is in* `Post-Deliberation phase`*, and all its related Analysis projects have transitioned to* `Post-Deliberation phase`*, then, if any changes have occured among solutions in those projects during the transition, the project should be switched back to* `Deliberation phase` *and the changes should be communicated to the project's team.* This rule ensures proper communication of new or adjusted solutions between teams of interrelated Analysis projects and allows a collaboration team to produce solutions that are not affected by possibly incorrect solutions produced by other teams. Similar strategies were adopted in agile software engineering methodologies, e.g., in SCRUM estimation game[1].

[1] http://scrummethodology.com/scrum-effort-estimation-and-story-points/



(a) Integrated relations between projects and actors

(b) Project types

Fig. 1. Projects in open source software engineering

2. *An engineering project design should be reviewed by an expert from a functionally dependent project. Moreover, it is preferable to assign an expert socially unrelated to the project team members.* This rule tries to avoid biased reviews by finding socially unrelated experts.
3. *In case of an expertise request, an appropriate expert should be socially connected to one of the project team members, or work on a related project.* This rule ensures faster expert onboarding.
4. *When starting an engineering project, a socially coherent team of qualified experts should be assembled, which has connections to members of related projects.* This rule tries to maximize probability of a project success by ensuring a good social environment in advance.
5. *An engineering project can be started, if at least one project it depends on has passed* `Design phase`. This rule defines a balance between total serialization of dependent projects `Design phases`, which results in a longer time-to-market, and total parallelization of `Design phases`, which results in more iterations.
6. `Design phase` *of a project cannot be finished until all projects it depends on pass* `Design phase`. This rule minimizes chances of potential rework and wasted efforts.
7. *If an engineering project is in* `Implementation phase`, *and any of the projects it depends on has switched back to* `Design phase`, *then the project should switch back to* `Design phase`. This rule covers possible redesign cases and ensures proper handling of late adjustments.
8. *All impediments in a project should be communicated to any engineer in every related project.* This rule ensures timely communication between project teams.

Let us consider the challenges that a developer faces when implementing the aforementioned rules. Based on the challenges, we further draw conclusions and identify the most important features that reflect the effectiveness of a coordination language and its underlying framework.

1. *Optimized horizontal integration of external collaboration projects.* The motivation scenario involves integration of many social networks and groupware products, such as MediaWiki, Subversion, LinkedIn, Facebook, and VersionOne. The developer should concentrate on the coordination logic, and not on how to extract the needed information from external sources. As APIs of collaboration platforms could not provide all the needed information in the right form, the framework needs to decouple the concepts perceived by the developer from representation and transformation issues and take care of the optimizing the data exchange seamlessly for the developer. Different authorization mechanisms and the necessity for identity mapping between entities coming from different sources makes integration even more complex. The coordination language should in turn support the unconditioned access to externally provided data in a manner that enables the optimization, and the language's semantics should reflect the nature of external APIs, i.e., consider distinct behavioral classes of APIs' methods (e.g., methods with and without side-effects).
2. *Condition-Action rules.* Rules 5, 6, and 7 take the declarative *condition-action* form, as opposed to more common *event-condition-action* rules, because the developer is interested in situations or patterns that need to be managed rather than in events that lead to these situations. When a condition depends on external data sources, problems of continuous checking and polling arise. Additionally, when a condition depends on time (e.g., escalation), timers get involved as well. These problems should be abstracted away from the developer and be handled by the framework, while the coordination language should support *condition-action* expressivity.
3. *Network context querying and processing.* Integration of groupware and social software enables social resource discovery and process coordination based on rich network context. Manipulations with network context, as it can be seen from most of the rules above, can be significantly simplified with quantifiers (as in Rules 5, 6, or 8), and disjunctions (as in Rule 3), as they naturally fit for expressing the coordination logic.
4. *Network context synchronization.* As depicted by Rule 1, when multiple related entities fulfill a rule, the action should be taken for all such entities simultaneously to avoid a situation when the action for one entity discards the condition for other related entities. Such synchronization issues should be handled at the framework level and be taken into account by the language design.

3 Related Work

In this section we examine existing coordination and orchestration languages with respect to the features outlined in the previous section. Table 1 summarizes

Table 1. Natively supported features in selected coordination languages

	Seamless integration	Condition-Action	Context queries	Context synchronization
Control-driven languages [1,12,13]	+/–	–	–	–
Linda-based languages [2,3]	–	+	+	–
Reactors [8]	–	+	+	–
CEP languages [20,21,23]	–	+	+	–
BPEL4Data (BEDL) [18]	–	+	–	–

the suitability of considered languages to network context-based coordination. The suitability of a language can be characterized as the amount of efforts a developer needs to spend on a task at hand. We therefore regard the native support of the aforementioned features, i.e., when no additional effort is needed for their realization.

Control-driven coordination and orchestration (workflow) languages based on messages (channels), such as BPEL [1], Orc [13], or Workflow Prolog [12] are specifically designed for integration of services like those in external APIs. They can also simulate network effects via messages or events, i.e., by notifying related processes. However, context querying using point-to-point messages would result in "chatty" communication, and context synchronization would require the implementation of complex protocols similar to two-phase commit. Also, support for integration is limited, as difference between methods w/o side-effects is not considered.

Data-driven (Linda-like [11]) coordination languages (for example, [2]) express coordination as dependencies between removal/reading and insertion of atoms from or into a shared space. However, groupware APIs are often assymetric and do not provide *insert/remove* operations for each *read* operation. It is therefore hard to align API method calls with the removal and insertion of atoms, because the actual changes made by API calls are not explicit and occur rather as side-effects. Basic Linda operators provide only limited expressivity of conditions expressing network context, unlike reactive Linda extensions [3] that introduce additional *notify* operation. Two coordination approaches are used in reactive extensions [3]: parallel (e.g., JavaSpaces, WCL) and prioritized (e.g., MARS, TuCSoN (ReSpecT)). In order to express network context synchronization, parallel reactions require implementing two-phase synchronization protocols, similarly to control-driven languages. Prioritized extensions make the implementation even more difficult by restricting the usage of coordination operators within reactions.

Reactors [8] is a coordination language where networks of reactors can be defined by means of relations. The behavior of reactors in the neighborhood is observed as sequences of their states, which can be queried with Datalog-based language, thus allowing the context querying. Also, Reactors eliminate the distinction between events and conditions. Reactors react to stimuli defined

as insertion or removal of relations. This is suitable for integrating RESTful APIs, but is limited to them, as many groupware APIs are coarse-grained and it is not intuitive to map insertion and removal of tuples to API calls. In general, reactors are executed concurrently and independently. Synchronous execution can only be achieved through a composition of reactors, which is not intuitive to implement.

Given that processes can publish their states as events, modern Complex Event Processing languages (e.g., [21]) can express conditions on network context using event correlation and predicates. However, representation of external data retrieved from request-response web APIs in the form of events is not intuitive. Moreover, the recursiveness [16] (See Rule 1) of collaboration processes can significantly complicate the definition of network context queries.

Typically, rules in Rule-based languages fire non-deterministically, thus complicating the network context synchronization. However, two notably different approaches here are: (i) to derive dependencies from postconditions (e.g., [23]), which in scope of external APIs integration might be not known, or not possible to define; and (ii) by explicit operators (e.g., [20]), which do not allow to specify dependencies based on relations between events.

In XML-based language BPEL4Data [18] processes can communicate via shared business entities, resembling thus a shared-space paradigm. Business entities are represented as XML documents. Simple conditions can be expressed as guards on Business entities using XPath/XQuery. However, it is not intuitive to describe network context querying, i.e., conditions on a graph of related XML documents. Synchronization between processes is achieved through additional processes and locks. Similarly to CEP languages, integration with BEDL requires representation of external data changes in the form of CRUDE notifications or invocations, which is not always intuitive.

As it can be seen, existing approaches partially support requirements outlined in the motivating scenario, but none of them provides a full spectrum of features necessary for efficient programming of coordination based on network effects.

4 Statelets Coordination Language

In this section we present Statelets, the coordination language designed for orchestration of activities in groupware and social software systems. The language natively supports all four features outlined in the previous section. However, native support of the *'Seamless integration'* feature requires additionally implementation of an extensible framework, conceptual design of which is discussed in the next section. The main building block of Statelets is *statelet* - a construct that corresponds to a state of a process and denotes coordination rules that should be fulfilled when the process resides in this state. Statelets do not completely describe collaboration processes, but rather are complementary reactions to workflows defined in groupware systems and human collaboration activities. A statelet consists of mainly two parts: a condition(s) that formally describes an anticipated situation and an action(s) which have to be undertaken if such a

situation is detected. Conditions are given in a form of *context queries* against the data integrated from external collaboration projects, and the actions are given as either *triggers* that correspond to external commands in collaboration software, or *yield* constructs that activate other statelets. All the data integrated from external sources by the framework is accessed as *relations* in the language. This allows the developer to easily design the coordination rules by seamlessly combining the relations originating from diverse platforms into single conditions.

4.1 Context Queries and Commands

Assymetric nature of many collaborative software APIs is reflected in Statelets as segregation of operations[2] to read (side-effect free) operations, i.e., queries, and modify operations, i.e., commands. Such segregation allows a programmer to specify what API methods are side-effect free and what are not, enabling thus the framework to treat them differently.

Queries. Read operations define data models, which in Statelets are represented as a unified hypergraph comprised of overlay networks. Even though the data model is defined by collaboration software adapters, additional relations may be integrated, (e.g., Core Relations Library), denoting side-effect free external computation. For example, querying the *Factorial(X, Factorial)* relation results in computation of a factorial by an integrated component. Also, additional virtual relations can be defined on top of the basic data model. For instance, a SocialRelation virtual relation below is defined by means of relations coming from Facebook and MySpace.

```
relation SocialRelation(User1, User2):
 Facebook.Friends(User1, User2) || MySpace.Friends(User1, User2);
```

Querying a hypergraph relation at runtime creates a *data stream*, i.e., a lazy sequence of records, which is gradually initialized by the framework with each set of vertexes matching the given relation found. Given that relations in hypergraph constitute predicates, data streams can be formed by expressions using the following binary operators based on the First-order logic:

- Operators `&&`, `||`, `not`, and `->` correspond appropriately to $\wedge$, $\vee$, $\neg$, and $\rightarrow$ first-order logic connectives with implicit existential quantification attached to all variables within the expression.
- Operators `=>`, `-!`, and `-x` correspond to conditional ($\rightarrow$) connector with implicit universal quantification over the variables present in the left part of the expression. Variables in the right part of the expression, that are not present in the left part, are quantified as $\exists$, $\exists!$, and $\neg\exists$ appropriately. Clearly, second and third operators can be expressed using the first one.

[2] http://martinfowler.com/bliki/CQRS.html

Basically, a query expression describes a pattern (a subgraph) within a hypergraph. Appropriately, a data stream resulted from evaluation of this query contains all occurrences of the pattern.

Queries in Statelets can be evaluated using `define` and `wait` operations:

- `define` operation simply evaluates a query expression and searches shared space hypergraph for pattern instances. Each pattern instance found along the hypergraph search is pushed into the data stream. If no instances are found, then `define` returns an empty data stream.
- `wait` operation continuously evaluates a query expression until at least one pattern instance is found. Therefore, `wait` operation always returns non-empty data stream.

For instance, if it is necessary to wait until all related to the project documents are completed, then we can use the following code snippet:

```
wait Related(Project, Document) => Status(Document, 'Completed');
```

Here a data stream is created that remains uninitialized until the condition is satisfied. However, if it is simply necessary to check if all related documents are completed, then the following code snippet can be used:

```
define Related(Project, Documents) => Status(Document, 'Completed');
```

Here an uninitialized data stream is created, which either is initialized with all related documents if all of them are completed, or is initialized as empty. A statelet can run many queries, getting thus many data streams. If query expressions within a statelet share variables, then resulting streams are joined by those shared variables.

Commands. Commands represent groupware API methods with side effects, for example, send an e-mail, or delete a document. Commands in Statelets are executed using `trigger` keyword:

```
trigger AssignReviewer(Document, Reviewer);
```

Commands in Statelets are used to process or handle records of data streams defined by query evaluations. If a data stream is not yet initialized, then a command is suspended until it is initialized (similar to lists with unbound dataflow tail [24]). However, if a data stream is empty, then the command is not executed at all. A command can be executed for `any` or for `every` record in a data stream, or for the whole collection of records. `Any` quantifier is a default quantifier, which is implicitly attached if no quantifiers are specified. Consider the example below:

```
trigger SendForReview(every Team, any Programmer, all Documents);
```

This reads as follows: send a list of Documents (all Documents) for a review to any Programmer in every Team.

4.2 Programming Coordination

Coordination is managing dependencies between activities. Apart of being able to express basic dependencies between human activities, Statelets also support network context-based coordination.

Dependencies between Activities. A statelet by itself describes precedence dependency: once completion of a human activity is registered in a shared space, a succeeding activity is triggered by a command. Statelets can be composed using `alternative` keyword expressing thus multiple different outcomes of a manual or automated activity. We exemplify usage of such composition in the use case scenarios. The statelet in the example below describes dependencies between design activity, project owner notification activity, and assignment of multiple experts activity:

```
statelet DesignPhase(Project):
{
 wait DesignDocument(Project, Document) && Status(Document, 'Completed');
 trigger NotifyProjectOwner(Project);
 define ExpertiseKeywords(Document, Keyword) && FindEngineers(Expert, Keyword);
 trigger Assign(every Keyword, any Expert, Project);
};
```

Dependencies between Processes. A process in Statelets is comprised of a sequence of statelets that produce each other by using `yield new` operation, i.e., a sequence of states. A process may reside in multiple orthogonal states, requiring thus presence of many statelets in parallel. Therefore, a statelet is technically a coroutine: it can produce multiple new statelets along its execution. Statelet by itself complements shared space hypergraph at runtime, simulating thus a relation. In other words, a statelet can query existence of other statelets in its neighborhood similarly to how it queries for existence of specific relations and nodes in a shared space hypergraph. A process in Statelets thus communicates with its neighborhood by changing its own state. In other words, observable behaviors of Statelets processes are sequences of *states*, rather then *messages*. This behavior was inspired by Cellular Automata [19], a popular abstraction for modeling complex behaviors in social and biological networks. If a statelet queries for the presence of another statelet, then such situation is treated by the framework as dependency, i.e., the assumption is that any actions triggered by a statelet can discard conditions of dependent statelets. Therefore, the framework ensures that actions of a statelet are triggered after conditions in dependent statelets are checked. Appropriately, if two statelets are mutually dependent, then the framework executes their actions simultaneously, allowing thus for expressing simultaneity dependencies, i.e., network context synchronization and collective actions (see Sec. 2). Lifetime of a statelet is bound to the data streams defined within it. A statelet is visible in shared space hypergraph until all its data streams are initialized. Once the statelet starts processing data streams by triggering actions, it becomes invisible to other statelets, i.e., queries being evaluated within `wait` operations of all other statelets will not consider presence of the relation correspondent to the statelet.

Let us consider an example: an engineering project can be started if design of all projects it depends on is finalized, and if at least one of them is in the implementation phase. The following code snippet implements this rule:

```
statelet DesignFinalizedPhase(Project):
{
 wait Depends(Project, DepProject) => (DesignFinalizedPhase(DepProject)
    || ImplementationPhase(DepProject));
 yield new ImplementationPhase(Project);
};
```

4.3 Feature Support and Prototype Implementation

All four features outlined in Sec. 2 are integral part of and natively supported by Statelets. Data streams and segregation of operations realize the horizontal integration feature. `Wait` operation enables *condition-action* rules. Implicit quantifiers in queries along with explicit quantifiers in commands allow for easy network context querying and processing. Statelet dependency solves the synchronization problem.

Statelets employ accustomed C-based syntax. Prototypes of the Statelets interpretor and the initial version of the language runtime are implemented in the functional programming language F#, and are publicly available for download[3].

The complete abstract syntax tree of the Statelets coordination language is provided below:

```
Quantifier Q ::= any | every | all
Constant C ::= boolean | number | string
 Identifier ID ::= string without spaces
Expression variables EVARS ::= ( ID | C | _ ) list
Command variables EPARS ::= ( Q ID | C ) list
Expression E ::= EVARS | (ID, EVARS) | (E && E) | (E || E) | (not E) | (E -> E)
   | (E => E) | (E -! E) | (E -x E)
VirtualRelation VR ::= (ID, ID list , E)
Statement S ::= define E | wait E | trigger ID EPARS | yield new ID EPARS
Statelet ::= (ID, ID list , S list )
```

5 Statelets Framework

In this section we present the conceptual architecture of the Statelets framework that enables horizontal integration of collaborative software systems. The focus of this paper is on the coordination language, therefore technical details are not provided. Figure 2 shows the high level design of the framework comprised of the following layers:

Connectors. Groupware APIs are diverse by their nature and employ distinct protocols. This requires creation of fine-tuned integration points, i.e., connectors. Connectors define supported relations and commands, and adapt object models of groupware APIs to fit Statelets semantic model. Connectors may support not only initialization of data streams corresponding to atomic relations, but also

[3] http://sourceforge.net/p/statelets/

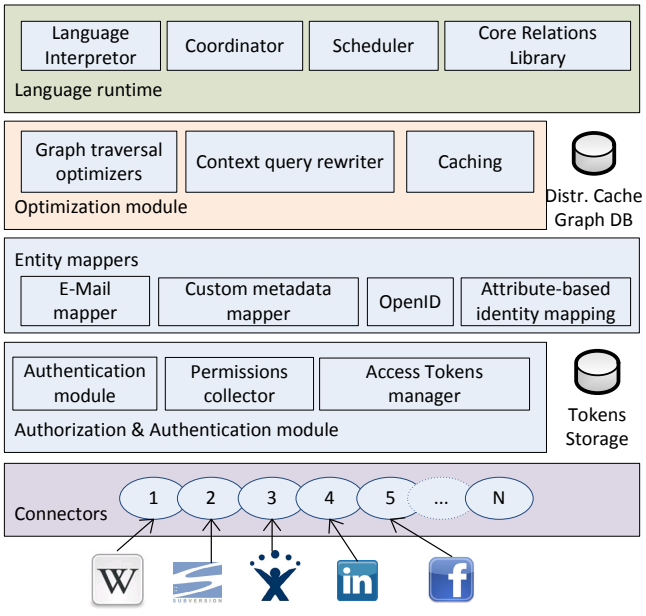


Fig. 2. Statelets framework architecture

interpretation of queries on relations in order to better utilize flexibility of APIs and improve efficiency.

Authentication and Authorization. User-centric APIs are designed for vertical composition [6], and often require authorization and authentication mechanisms with direct user involvement (e.g., OAuth 1.0/2.0). This complicates traversal of social graphs, and imposes needs to store and maintain certificates, application and user tokens, or even credentials. Moreover, a mechanism to update or collect new tokens should be present as well.

Entity Mapping. Many user accounts and entities map to the same entity in the real world. For instance, users usually have different accounts per each collaboration tool they use, and two files in different tools may represent the same research paper. Typical approaches to entity mappings [14] are attribute-based identity, by e-mail address, by custom metadata, or even direct mappings (e.g., based on Facebook Open Graph or OpenID).

Optimizations. Authentication and authorization mechanisms together with identity mappings algorithms may introduce high latency. Additionally, some data in social networks, like a friendship connection, or a user profile, change rarely. This introduces unnecessary overhead for queries with existential quantifiers, i.e., 'find any socially related expert in given area'. In this case, caching and heuristic approaches may bring substantial value.

Language Runtime. The language interpretor is responsible for code parsing and interpretation of the language semantic model. The scheduling component

is responsible for polling graphs of artifacts and user profiles. The coordination component is responsible for enforcing dependencies between activities and processes at runtime.

The multi-layer design decouples integration and optimization issues from the coordination logic. The developer therefore only operates with entity abstractions and is not required to comprehend technical details of data access, whereas the other layers are handled by appropriate integration experts.

6 Use cases

This section demonstrates the implementation of two process types considered in Sec. 2, namely Analysis and Engineering projects. The use cases exemplify main language features and implementation of coordination based on network effects. More use-cases can be found online[3].

6.1 Analysis Projects

MediaWiki engine used in Wikipedia is used as an underlying groupware platform. Typically, work on wiki pages is coordinated by non-functional attributes, for example, 'Category:All articles with unsourced statements'. Similarly, we add a special marker category which is used to denote `Post-Deliberation phase` of a project. Two analysis projects are considered to be related, if one of the project wiki pages contains a link to a wiki page from the other project. Synchronization between related projects is achieved in two steps: (i) residing in the Post-Deliberation phase, a process waits until all related processes switch to the Post-Deliberation phase; (ii) all changes made in related projects since last synchronization are communicated to every team member in related projects, and related projects switch back to the Deliberation phase simultaneously.

```
statelet AnalysisProject.Deliberation(WikiPage, Timestamp):
{
 wait Wiki.Categories(WikiPage, "PostDeliberation");
 yield new AnalysisProject.PostDeliberation(WikiPage, Timestamp);
};

statelet AnalysisProject.PostDeliberation(WikiPage, Timestamp):
{
 wait
  ((Wiki.Links(WikiPage, RelatedPage) => AnalysisProject.PostDeliberation(RelatedPage, _))
  -> Wiki.Revisions(RelatedPage, _, RelRevTimestamp))
  && >(RelRevTimestamp, Timestamp) && System.DateTime.Now(now);
 define Wiki.Revisions(WikiPage, Contributor, _);
 trigger Wiki.EmailUser(every Contributor, every RelatedPage, all RelRevTimestamp);
 trigger Wiki.DeleteCategory(WikiPage, "PostDeliberation");
 yield new AnalysisProject.Deliberation(WikiPage, now);
}
alternative
{
 wait WikiPage -x Wiki.Categories(WikiPage, "PostDeliberation");
 yield new AnalysisProject.Deliberation(WikiPage, Timestamp);
};
```

This use case exemplifies simplicity of network context synchronization and collective actions implementation in case of recursive collaboration processes.

6.2 Engineering Projects

To save space, we exemplify only expertise discovery in social neighborhood. The algorithm combines two ideas: (i) try to find a reviewer from a related project, which is not socially related to any of the project team members; (ii) try to find any reviewer who has appropriate expertise. In this example, social context is retrieved from Facebook, LinkedIn, and Subversion (two engineers are socially related if they committed to the same project in subversion). Project data is retrieved from the VersionOne groupware. Subversion and VersionOne are depicted in the code snippet as `SVN` and `V1` respectively.

```
relation SVN.Related(User1, User2):
 SVN.Logs(Path, User1, _, _, _) && SVN.Logs(Path, User2, _, _, _);

relation SocialRelation(User1, User2):
 SVN.Related(User1, User2) || Facebook.Friends(User1, User2);

statelet EngeneeringProject.InProgress(Story):
{
 wait V1.Attribute(Story, "Status", "Completed");
 yield new EngineeringProject.ImplementationFinished(Story);
}
alternative
{
 wait
  V1.Attribute(Story, "Status", "Review") && not V1.Relation(Story, "Reviewer", _)
  && V1.Relation(Story, "Developer", Dev)
  && V1.Relation(Story, "FunctionalRelation", RelStory)
  && V1.Relation(RelStory, "Developer", RelDev)
  && LinkedIn.Profile(RelDev, Profile) && ExpertiseFits(Profile, Story)
  && (not SocialRelation(Dev, RelDev) || RelDev);
 trigger SetRelation(Story, "Reviewer", any RelDev);
 yield new EngineeringProject.InProgress(Story);
};
```

The use case exemplifies implementation of such advanced activities as location of socially connected experts, unbiased reviewers, and so on. The use case also shows benefits arising from horizontal composition of social networking sites.

7 Conclusions and Future Work

This paper proposes a novel coordination language for network context-based coordination, and demonstrates its suitability through use cases. Compared to existing approaches, our contribution provides a full spectrum of features that are crucial for network context furnishing and coordination based on it. We have shown that these features are necessary for an effective coordination of social collaboration processes. However, at present the language is in its inception phase, and does not support advanced features (e.g., hierarchical composition of Statelets) for expressing more complex and large-scale coordination problems than those exemplified in the use cases. Therefore, our future work includes further advancement of the Statelets coordination language, and design and development of various techniques aiming at optimized integration of various groupware and social networking sites APIs. Although Statelets was designed with the focus on collaboration, we do not exclude its applicability in other areas.

References

1. Arkin, A., Askary, S., Bloch, B., Curbera, F., Goland, Y., Kartha, N., Liu, C.K., Thatte, S., Yendluri, P., Yiu, A.E.: Web services business process execution language version 2.0 (May 2005)
2. Banâtre, J.-P., Fradet, P., Le Métayer, D.: Gamma and the Chemical Reaction Model: Fifteen Years After. In: Calude, C.S., Păun, G., Rozenberg, G., Salomaa, A. (eds.) Multiset Processing. LNCS, vol. 2235, pp. 17–44. Springer, Heidelberg (2001)
3. Busi, N., Zavattaro, G.: Prioritized and Parallel Reactions in Shared Data Space Coordination Languages. In: Jacquet, J.-M., Picco, G.P. (eds.) COORDINATION 2005. LNCS, vol. 3454, pp. 204–219. Springer, Heidelberg (2005)
4. Chwe, M.S.Y.: Communication and coordination in social networks. Review of Economic Studies 67(1), 1–16 (2000)
5. Dustdar, S., Bhattacharya, K.: The social compute unit. IEEE Internet Computing 15(3), 64–69 (2011)
6. Dustdar, S., Gaedke, M.: The social routing principle. IEEE Internet Computing 15(4), 80–83 (2011)
7. Dustdar, S.: Caramba a process-aware collaboration system supporting ad hoc and collaborative processes in virtual teams. Distributed and Parallel Databases 15, 45–66 (2004)
8. Field, J., Marinescu, M.C., Stefansen, C.: Reactors: A data-oriented synchronous/asynchronous programming model for distributed applications. Theoretical Computer Science 410(23), 168–201 (2009)
9. Florijn, G., Besamusca, T., Greefhorst, D.: Ariadne and HOPLa: Flexible Coordination of Collaborative Processes. In: Ciancarini, P., Hankin, C. (eds.) COORDINATION 1996. LNCS, vol. 1061, pp. 197–214. Springer, Heidelberg (1996)
10. Galeotti, A., Goyal, S., Jackson, M.O., Vega-Redondo, F., Yariv, L.: Network games. Review of Economic Studies 77(1), 218–244 (2010)
11. Gelernter, D., Carriero, N.: Coordination languages and their significance. Commun. ACM 35, 97–107 (1992)
12. Gregory, S., Paschali, M.: A Prolog-Based Language for Workflow Programming. In: Murphy, A., Vitek, J. (eds.) COORDINATION 2007. LNCS, vol. 4467, pp. 56–75. Springer, Heidelberg (2007)
13. Kitchin, D., Cook, W., Misra, J.: A Language for Task Orchestration and Its Semantic Properties. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 477–491. Springer, Heidelberg (2006)
14. Ko, M.N., Cheek, G., Shehab, M., Sandhu, R.: Social-networks connect services. Computer 43(8), 37–43 (2010)
15. Liptchinsky, V., Khazankin, R., Truong, H.L., Dustdar, S.: A novel approach to modeling context-aware and social collaboration processes. In: The 24th International Conference on Advanced Information Systems Engineering (CAiSE 2012) (2012)
16. Martinez-Moyano, I.: Exploring the dynamics of collaboration in interorganizational settings. In: Creating a Culture of Collaboration: The International Association of Facilitators Handbook, vol. 4, p. 69 (2006)
17. McDonald, D.W., Ackerman, M.S.: Just talk to me: a field study of expertise location. In: Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work, CSCW 1998, pp. 315–324. ACM, New York (1998)

18. Nandi, P., Koenig, D., Moser, S., Hull, R., Klicnik, V., Claussen, S., Kloppmann, M., Vergo, J.: Data4bpm, part 1: Introducing business entities and the business entity definition language (bedl) (April 2010)
19. Neumann, J.V.: Theory of Self-Reproducing Automata. University of Illinois Press, Champaign (1966)
20. Núñez, A., Noyé, J.: An Event-Based Coordination Model for Context-Aware Applications. In: Lea, D., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 232–248. Springer, Heidelberg (2008)
21. Plociniczak, H., Eisenbach, S.: JErlang: Erlang with Joins. In: Clarke, D., Agha, G. (eds.) COORDINATION 2010. LNCS, vol. 6116, pp. 61–75. Springer, Heidelberg (2010)
22. Powell, A., Piccoli, G., Ives, B.: Virtual teams: a review of current literature and directions for future research. SIGMIS Database 35, 6–36 (2004)
23. Shankar, C., Campbell, R.: A policy-based management framework for pervasive systems using axiomatized rule-actions. In: Proceedings of the Fourth IEEE International Symposium on Network Computing and Applications, pp. 255–258. IEEE Computer Society Press, Washington, DC (2005)
24. Van Roy, P., Haridi, S.: Concepts, Techniques, and Models of Computer Programming. The MIT Press (February 2004)
25. Zhang, Y., Bolton, G.E.: Social Network Effects on Coordination: A Laboratory Investigation. SSRN eLibrary (2011)