A Domain Specific Language for the Development of Collaborative Systems

Luis Mariano Bibbo [1] and Diego García [1] and Claudia Pons [1,2]

[1] *LIFIA, Faculty of Informatics, National University of La Plata*
[2] *Facultad de Tecnología Informática, Universidad Abierta Interamericana*
Buenos Aires, Argentina.
{lmbibbo, dgarcia, cpons}@sol.info.unlp.edu.ar

Abstract

Domain-Specific Languages (DSLs) are high level languages defined for combining expressivity and simplicity by means of constructs which are close to the problem domain and distant from the intricacies of underlying software implementation constraints. This paper presents a language to graphically document the analysis and design decisions embodied in Collaborative System development. The language was designed as a conservative extension of the UML and it enables the application of the MDD approach to the development of such systems.

1. Introduction

Collaborative software is software designed to provide support to people engaged in a common task to achieve their goals [1]. Collaborative software is the basis for computer supported cooperative work (CSCW) [2], cooperative learning (CSCL) and cooperative game-play (CSGP). The research on computer-supported collaboration is focused on the possibilities and effects of technological support for groups, organizations, communities and societies involved in these collaboration processes. Nowadays, technological improvements in computer networks, especially on the Internet, allow more and more people to connect online to enjoy better collaborative applications in the area of business as well as entertainment.

The construction of collaborative software is a very complex task due to the fact that such systems involve numerous intensely-interactive actors that are dispersed over different locations. The adoption of a process that guides the construction of such systems is crucial because it encompasses repeatable practices and techniques that organize software development and favor software quality. Different processes vary in when to perform each development activity, on which part of the system these activities are performed, and the rigor and formality of the artifacts to be built.

In the context of software engineering, the model-driven development approach (MDD) [3] [4] [5] has emerged as a paradigm shift from code-centric software development to model-based development. Such an approach promotes the systematization and automation of the construction of software artifacts. Models are considered as first-class constructs in software development, and developers' knowledge is encapsulated by means of model transformations. The essential characteristic of MDD is that software development's primary focus and work products are models. A model is an abstraction of the reality that helps on mastering a large and complex system that cannot be comprehended in its entirety. A model is a semantically closed abstraction of a system, in the sense that it fully describes the system at the chosen level of precision and from a particular viewpoint. Its major advantage is that models can be expressed at different levels of abstraction and hence they are less bound to any underlying supporting technology. A key premise for MDD is that models are useful beyond documentation. They can also be used to generate other work products automatically.

To successfully apply the model-driven approach to the development of collaborative systems we need in first place to be able to create abstract models of the system. Such models can be written in plain UML [6]; however this task will be certainly overwhelming for developers due to the deep gap that exists between the UML semantics and the collaborative systems domain's semantics. Therefore, we need to count with a languages tailored to precisely match this particular domain's semantics, in contrast to general-purpose ones that do not express in a direct manner the domain concepts and intent. The domain specific language constructs [20] will allow for the abstract conceptualization of the domain in a precise but concise and simple way. On the other hand, such language will enable the automation of the construction

of the concrete software artifacts from the abstract models.

There has been a significant amount of theory built around the development of languages (mostly UML extensions) for specifying systems involving some kind of collaborative activities, such as the language to model hypermedia and distributed systems presented in [7], the notation to model shared data in collaborative systems described [8], the language for specifying distributed system introduced in [9], the UML extension named AUML [10] for describing communication protocol between agents and the extension for interactive applications defined in [11] among others. Although these languages offer specific notation for modeling a number of particular features of collaborative systems, no one of them covers the domain in an exhaustive way, leaving aside several concepts. Besides, those languages are not intended to serve as the step stone for applying the MDD philosophy in the development of collaborative systems.

In this paper we present the definition of a language that deals with the precise description of systems in this particular domain. The language is defined as an extension of the UML by using the mechanism of metamodeling that is a quite commonplace technique in the syntax specification of DSLs.

The paper is structured as follows; Section 2 describes the collaborative systems domain in detail. Section 3 explains the rational for our language design process. Section 4 presents the formal definition of the domain specific language. Section 5 gives a concrete example of application of the proposed language. Finally, section 6 recaps the contribution of this paper and analyzes its relation to similar works.

2. Collaborative systems domain

Collaborative software facilitates and manages group activities. Collaborative software is software designed to provide support to people engaged in a common task to achieve their goals [1]; it provides a friendly interface to a shared environment. In an effective collaboration the main function of the participants is to alter a collaboration entity. The collaboration entity is an adaptable object; examples include the development of an idea, the creation of a design, and the achievement of a shared goal. Typical collaboration technologies include document management, threaded discussions, audit history, context awareness, change notification and other mechanisms designed to capture the efforts of the group of participants into a managed shared environment.

Collaborative software can be divided into three categories depending on the level of collaboration: communication level, collaboration level and co-ordination level. Communication level can be thought of as unstructured interchange of information, for example a phone call or a chat discussion. Collaboration level refers to interactive work toward a shared goal. Brainstorming and voting are examples of this collaborative activity. Co-ordination level refers to complex interdependent work toward a shared goal. A good metaphor for understanding this is to think about a sports team; everyone has to contribute the right play at the right time as well as adjust their play to the unfolding situation - but everyone is doing something different - in order for the team to win.

This last level of collaborative software is the most interesting since it subsumes the other two levels. Examples include: electronic calendars that schedule events and automatically notify and remind group members; project management systems that schedule, track, and chart the steps in a project as it is being completed; workflow systems which collaborative manage the tasks and documents within a knowledge-based business process; knowledge management systems that collect, organize, manage, and share various forms of information; prediction markets that let a group of people predict together the outcome of future events; extranet systems that collect, organize, manage and share information associated with the delivery of a project; social software systems which organize social relations of groups; online spreadsheets that share structured data and information.

With the aim of building abstract models of this kind of system, we need to count with a languages customized to fit this particular domain's semantics. In the next section we define the abstract syntax of such a language. It is defined as an extension of the UML by using the mechanism of metamodeling that is a quite commonplace technique in the specification of the syntax of domain specific languages.

3. Language design process

Our task as language designers was to choose the most suitable structure for the language and define it into a metamodel. For accomplishing this task we mainly cope with two issues: first of all we should find the sources for identifying the modeling concepts that are involved in collaborative systems design. Then, we should decide starting the language definition either from scratch or by modifying existing languages.

With respect to the source for discovering modeling concepts, we considered two options: the first option consists in using modeling concepts that originate from

the code while the second option consists in taking non implementation concepts. We opted for the second option, that is to say we focus more on the problem domain than its code. We believe that better productivity can be achieved if the modeling language is close related to the problem domain concepts rather than code, because the problem domain concepts are usually already known and used by non experts and have established semantics in place. The identification of the modeling concepts is normally highly iterative and may need to be repeated several times. The identification of all the concepts immediately is quite unfeasible and therefore it helps to define languages in stages and try them out by making example models. The number of iterations depends on the size of the domain, its stability, availability of domain knowledge, and the experience of the language definers. The proposal in this paper is still under development, it will require additional iterations until completion.
Finally, regarding the decision between starting the language definition from scratch or by modifying existing languages, we opted for extending the UML language in order to reuse several concepts such as Classifiers, Packages and State Machines.

4. Collaborative software system language (CSSL)

A model of a collaborative system is a description of the system written in a well-defined language, which is suitable for automated interpretation by a computer. The most commonplace technique for the specification of the syntax of graphical languages is called *metamodeling*. A metamodel defines what elements can exist in a model. For example, the UML metamodel defines that we can use the concepts "Class," "State", "Package", etc., in a UML model. Metamodels are also graphical models, thus a metamodel itself must be written in a well-defined language. This language is called a metalanguage. In theory there are an infinite number of layers of model-language-metalanguage relationships, but the standards defined by the OMG use four layers, called M0, M1, M2, and M3.

Layer M0: The Instances. At the M0 layer there is the running system in which the “real" instances exist.

Layer M1: The Model of the System. The M1 layer contains models, for example, a UML model of a software system. There is a definite relationship between the M0 and M1 layers. The concepts at the M1 layer are all categorizations of elements at the M0 layer. Likewise, each element at the M0 layer is always an instance of an element at the M1 layer.

Layer M2: The Model of the Model (the Metamodel). The elements that exist at the M1 layer are themselves instances of classes at the M2 layer, while the concepts at the M2 layer are all descriptions of elements at the M1 layer. M2 contains concepts needed to create models. The models that reside in M2 are called metamodels.

Layer M3: The Model of the Metamodel (the Metametamodel). Along the same line, we can see an element at the M2 layer as being an instance of an element at yet another layer, the M3 or metameta layer. Again the same classification-instantiation relationship exists between both layers. Within the OMG, the MOF [12] is the standard M3 language.

Thus, all modeling languages are instances of the MOF; their definitions (or metamodels) live in the M2 layer. For example, the UML metamodel and the OCL metamodel [13] live in this layer. On the other hand, the UML offers a mechanism to extend and adapt its metamodel to the specific needs of a software platform (e.g., J2EE) or to the particularities of an application domain (e.g., Real Time Systems, Collaborative Systems), so that when creating a new modeling language we do not need to start from scratch. UML metaclasses can be specialized to reflect the specific concepts. After that, the specialized metamodel can be easily implemented by using the profile mechanism of UML. Taking advantage of this mechanism, in the next sections we present an extension of the UML metamodel designated to define a domain specific language for Collaborative Software Systems, called CSSL. The metamodel of CSSL is structured in 3 packages, as illustrated in figure 1.

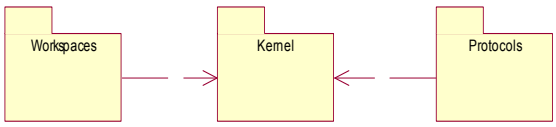

Figure 1. Overall Packages in the CSSL metamodel

Kernel: this package contains all the concepts and abstractions that serve as basis for the rest of the metamodel packages, such as SharedObject and CollaborationRole.

Workspaces: It includes the modeling elements for describing the structural features of the collaborative system, such as Workspace.

Protocols: this package holds the modeling elements for describing the dynamics of the collaborative systems, such as StateMachine.

These packages import metaclasses form the UML2 metamodel [14] that is an EMF-based implementation

of the Unified Modeling Language (UML) 2.x metamodel for the Eclipse platform. In the following figures the original metaclasses of the UML2 are displayed in light-gray while the specific metaclasses that were added to this metamodel are highlighted in dark-gray. Due to space limitations, only a few elements of each package are described in detail.

Kernel

The Kernel package contains the basic artifacts that serve as the foundation for the remaining packages. The main metaclasses in this package are: Tool, SharedObject and CollaborationRole. The package structure is depicted in Figure 2.

- **Collaboration Role:** an instance of this modeling artifact denotes the role that an individual can play in the system. Each role is defined by specifying its qualifications (or skills), its responsibilities (which actions should perform) and the shared objects and tools that the role is allowed to use.

Properties:

qualification: String It specifies the skills that the role possess.

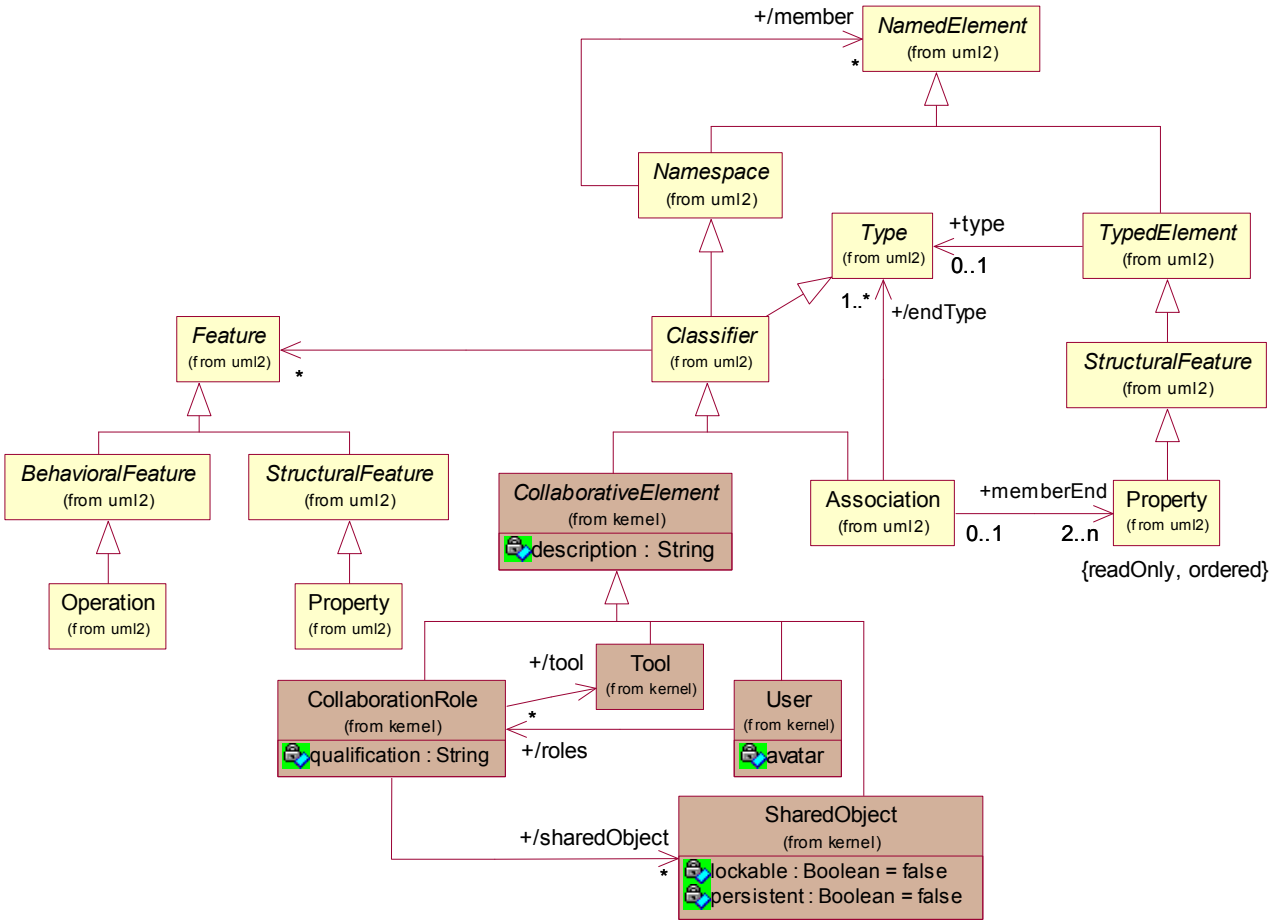

Figure 2. Kernel Package

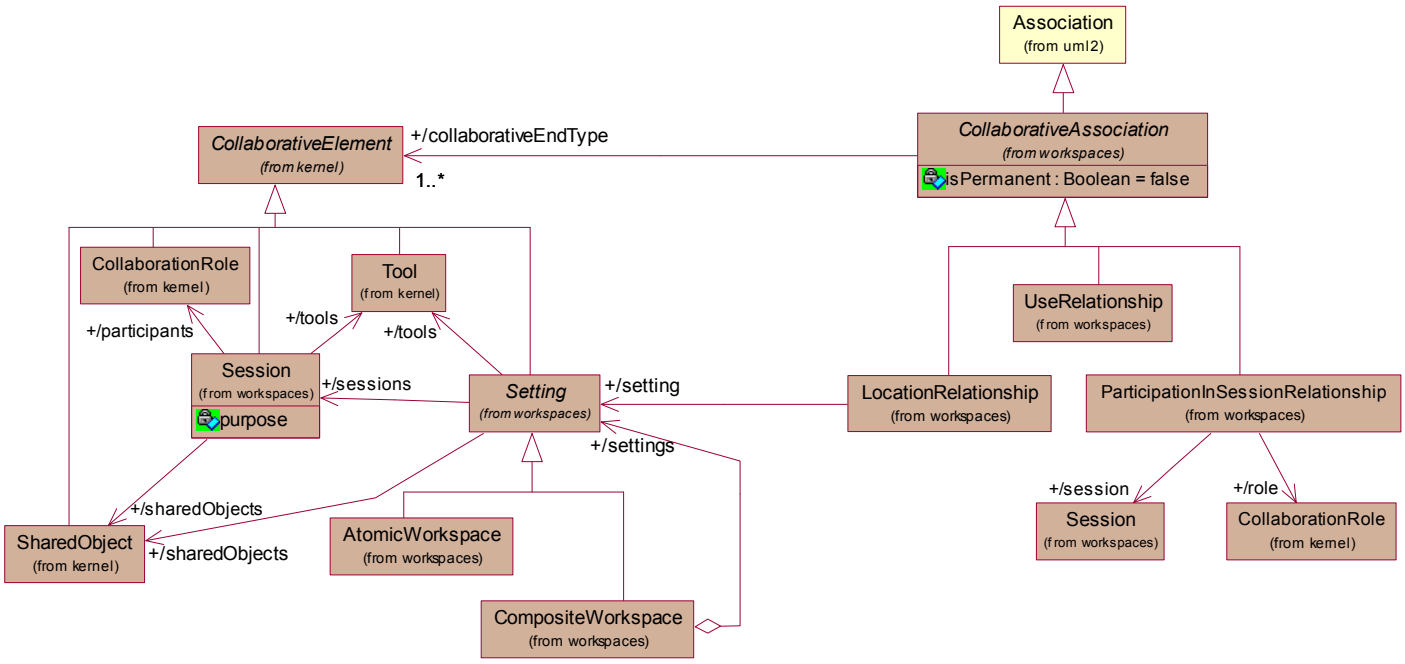

Figure 3. Workspaces Package

responsibilities: Operation It describes the set of responsibilities of the role. Responsibilities are modeled as UML Operations.

/SharedObjects: SharedObject It specifies the set of objects that the role possesses. It is a property derived from Classifier./AllAttributes: *self.allAttributes->select(p| p.type.oclIsKindOf(SharedObject))*.

/tools: Tool It denotes the set of tools that the role is allowed to use. It is a property derived from Classifier./AllAttributes: *self.allAttributes -> select (p | p.type.oclIsKindOf(Tool))*.

Workspaces

The Workspaces Package contains the artifacts that can be used for modeling the structural aspects of the environment where the collaborative activity takes place. In particular the package defines the workspace artifact together with the different kind of structural relations that can be established between workspaces and other collaborative elements. The Workspaces Package content is illustrated in figure 3.

▪ **Setting:** This metaclass denotes the virtually physical place where the collaboration takes place. Inside a setting there are collaborative tools and objects that are used by the roles in order to accomplish their tasks. It is an abstract metaclass, its subclasses are AtomicWorkspace and CompositeWorkspace.

Properties:

/sessions: Session It represents the sessions that can take place in this workspace. It is a property derived from Classifier./AllAttributes: *self.allAttributes -> select (p | p.type.oclIsKindOf(Session))*.

/tools: Tool It represents the tools that are available into the workspace. It is a derived property.

/sharedObjects: SharedObject It specifies the set of shared objects that are located into the workspace. It is a property derived from Classifier/AllAssociations: *self.allAssociation->select (p | p.LocationRelationship) ->collect(p:LocationRelationship| p./collaborativeEndType) -> flatten() -> select (c | c.oclIsKindOf(SharedObject))*

▪ **AtomicWorkspace:** this metaclass represents a simple setting (without sub-settings).

Constraints

[1] An atomic workspace can not contain any sub-workspaces in its interior. *self./member -> select (p | p.oclIsKindOf(Setting)) -> isEmpty()*

▪ **CompositeWorkspace:-** A composite workspace is an arrangement of smaller workspaces. In this way complex structures can be created by combining simpler workspaces.-

Properties:

/settings: Setting This property denotes the sub-settings that integrates the composite workspace. It is property derived from Namespace./member: *self./member -> select (p | p.oclIsKindOf(Setting))*.

▪ **ColaborativeAssociation:-** this is an abstract metaclass. It represents the different kind of associations that can be established between elements. Its subclasses are:

LocationRelationship: it indicates a location relationship between a Setting and a CollaborativeElement (for exammple the tool A is located in the workspace B).

UseRelationship: it indicates a use relationship between a Setting and a CollaborativeElement (for exammple the tool A can be used in the workspace B).

ParticipationInSessionRelationship: it indicates a relationship between a Role and a Session (for exammple the role A participates in the session B).

▪ **Session:** a session is a period of information exchange between two or more participants or between the computer and a user. A session is set up at a certain point in time, and turn down at a later point in time. A session has a particular goal. A session typically contains shared information, meaning that it needs to save some information about its characteristics in order to make it available during the collaboration. For example the information about who the users that are working in the particular session are, which tools are available during the session, etc.

Properties:

purpose: String This property specifies the goal of the session.

/shared objects: Shared Object This is the list of shared objects that are available in this session. It is a property derived from Classifier./AllAttributes: self.allAttributes -> select(p| p.ocl.type.oclIsKindOf(SharedObject))

/tools : Tool This is the list of tools that are can be used in this session. It is a property derived from Classifier./AllAttributes : *self.allAttributes->select(p| p.type.oclIsKindOf(Tool))*.

/participants: Role This property indicates which participants can participate in the session.

Protocols

The Protocols package defines a set of concepts that can be used for modeling discrete behavior through finite state-transition systems. These modeling elements express the behavior of the collaborative system. Here the main modeling artifact is a refined version of the UML State Machine that is composed of states and transitions. The goal of a protocol is to define, organize and guide the social process that should be carried out by the individuals in the system. Each protocol establishes which actions the roles are allowed to carry out in each state of the system. Also the protocol specifies which tools and objects are available in each state. The package content is described in figure 4.

▪ **Session Transition:** a transition represents the change from one state to another when some conditions are fulfilled. Those conditions can be either actions performed by a role (e.g. the speaker ends the brainstorming session) or events such as reaching a deadline.

Properties:

source: Vertex It denotes the origin of the transition (in general, the state of the system before the firing of the transition).

target: Vertex It specifies the destination of the transition (in general the state of the system after the firing of the transition).

trigger: Trigger It specifies the cause of the transition firing. It could be a behavior performed by a role or it could be an external event such as reaching a deadline)

guard: Constraint It is the Boolean condition that should be fulfilled so that the transition becomes enabled.

role: CollaborationRole [0..1] The role is the individual that performed the action that fired the transition. (if any).

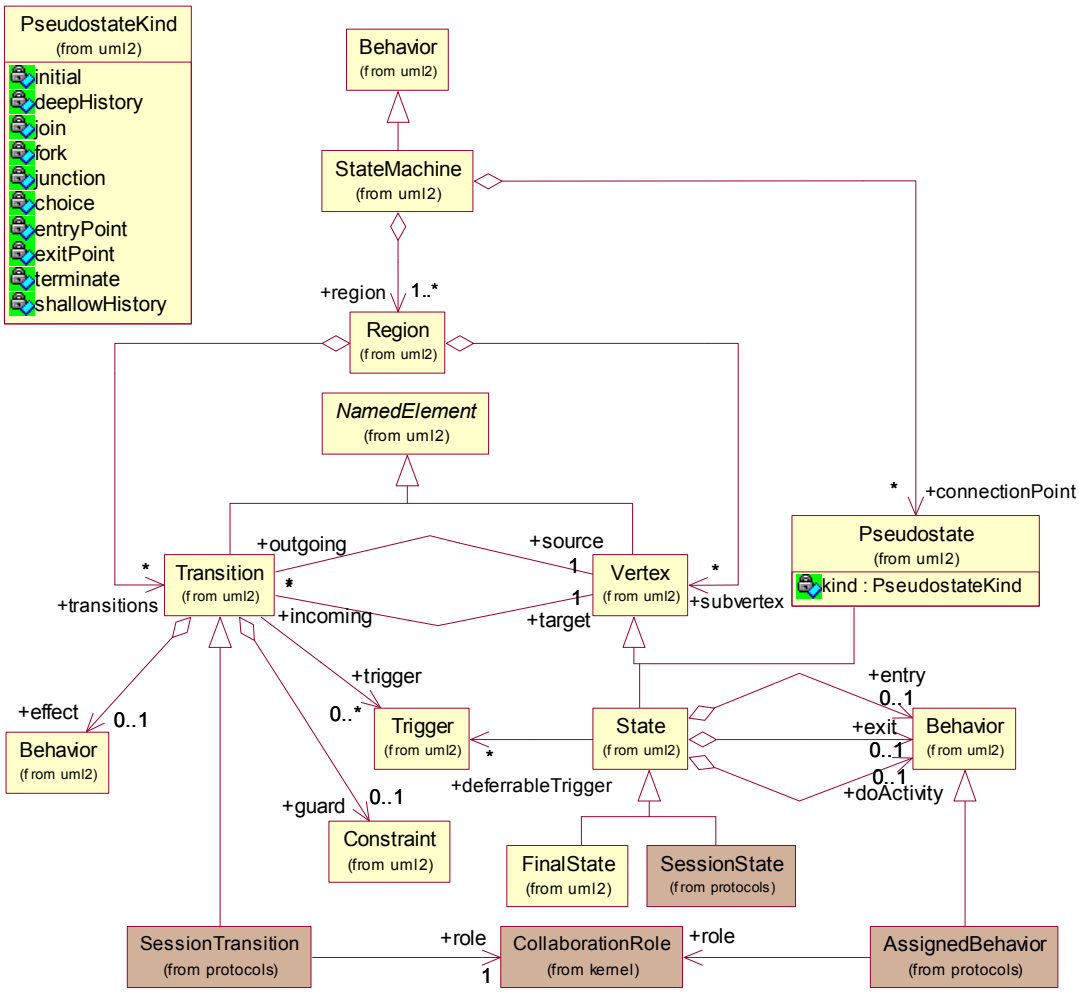

Figure 4. The Protocols Package

5. Case study

We present a design example of an application for brainstorming and group decision-making. Group decision-making software is designed to facilitate the work of groups, group decision-making and brainstorming. The case study will focus primarily on a simplified version of a real-time web application for brainstorming. The typical setup involves participants that are at their own computer connected to the system. The speaker presents the subject to discuss by using video conference tool and the facilitator gives a simple training on how the system works. Then the facilitator - typically hired to conduct the meeting - organizes the questions, polling and information gathering tasks. The participants have more involvement in the ideas gathering session. They add new ideas, organize other people's ideas and vote them to reach a final set of ideas as a conclusion of the meeting.

For this basic version of the system we find the following elements (for space limitations, only a few elements are described in detail):

i) Collaborative roles: The different roles in the system are Participant, Speaker and Facilitator. The CollaborationRole template (see figure 5) allows the developer to specify the name of the role, its description, qualifications, and the list of responsibilities, tools and shared object, as described in the CSSL metamodel.

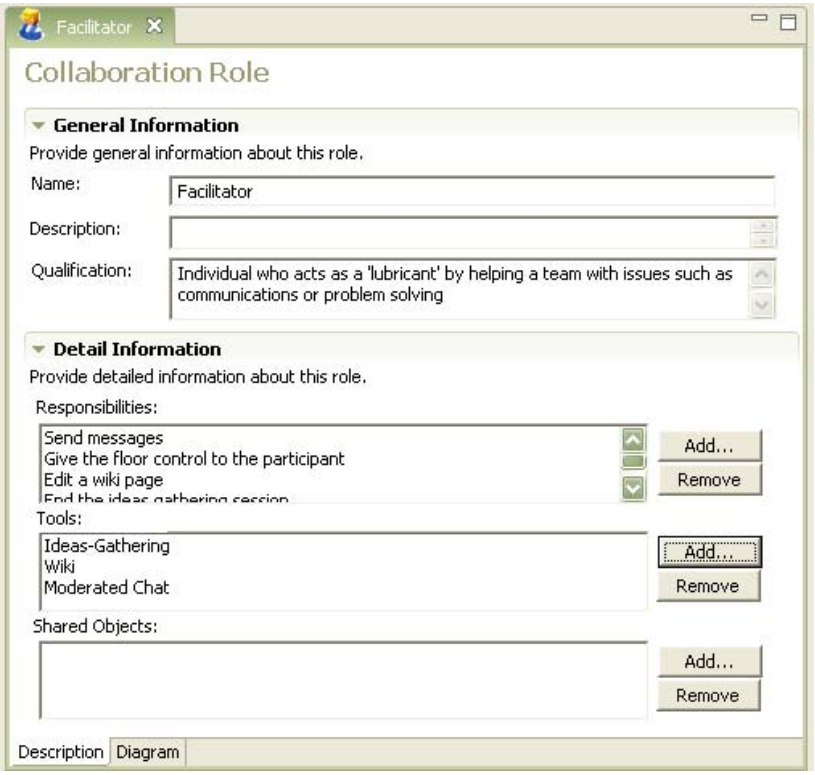

Figure 5. Concrete syntax of the CollaborationRole Metaclass

For example, the figure 5 shows the CSSL construct describing the Facilitator role. The list of responsibilities includes: start a session, send message, give the floor control to the participant, edit a wiki page, etc. The tools that this role can use are the following: the ideas gathering tool, the wiki, and the moderated chat. Additionally the language offers a more diagrammatic view of the connections between each role and its properties as it is displayed in figure 6. This view is automatically derived.

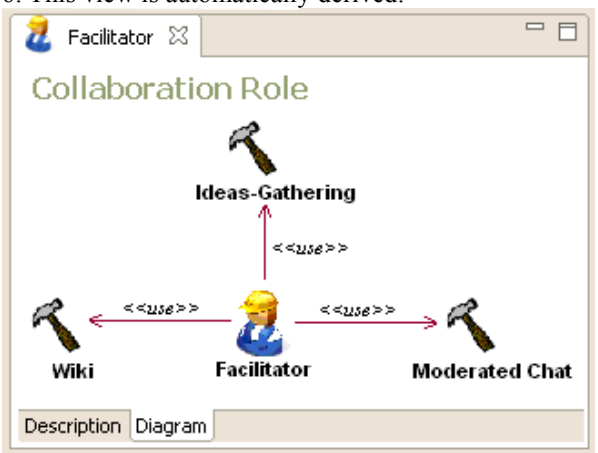

Figure 6. Alternative view of the usage relationship between roles and tools

ii) Workspaces: the system has two workspaces:

- the **Discussion Table**. In this workspace the participants discuss ideas and collect them by using the Ideas Gathering tool. Generally the table is designed for small group of six to ten participants. They can also use the chat tool to communicate with each other and the voting tool as a decision-making mechanism.
- the **Auditorium.** In this workspace the brainstorming process starts. The speaker presents the subject to discuss and the facilitator gives a simple training about the system. At the end of the brainstorming process the Conclusion session is carried out also in the auditorium. This workspace is designed to contain a bigger group of user. In this workspace the available tools are the broadcast video conference tool and the chat system to answer the participant's questions.

Figure 7 illustrates the CSSL template describing the Auditorium. In addition, figure 8 presents a graphical overview of the relationships between the workspace and the available tools, the shared objects and the sessions that take place inside the workspace. This view is automatically derived form the template.

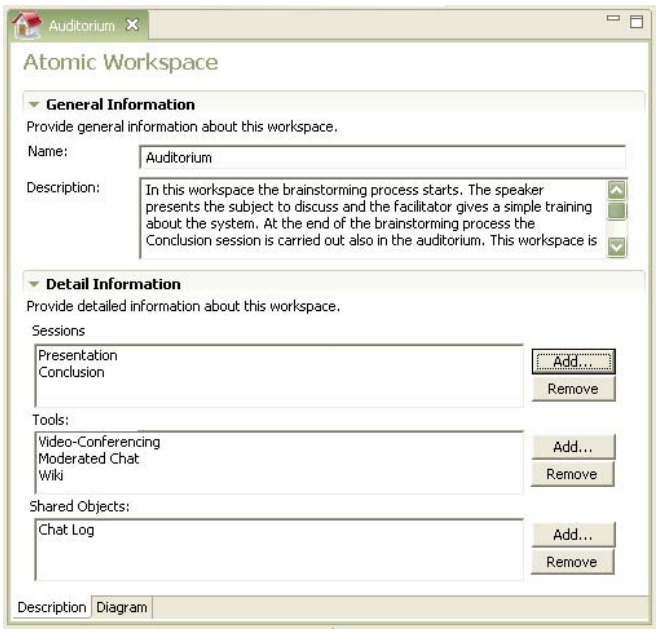

Figure 7. Concrete syntax of the Atomic Workspace metaclass

iii) Protocols: respect to the social process, in our case study we identify one main process - the Brainstorming Process – composed of three sessions that take place in a sequential order: the Presentation Session, the Collecting Ideas Session and the Conclusion Session. Figure 9 displays the overall protocol of the Brainstorming Process. Figure 10 shows the description of one of these sessions using the concrete syntax of CSSL. Internally each session has its own protocol. As example, the Figure 11 describes the protocol of the Presentation session.

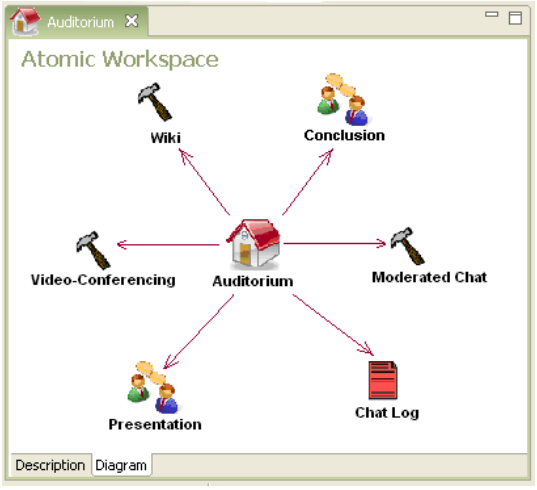

Figure 8. Derived view of the relationships between workspaces, tools, objects and sessions

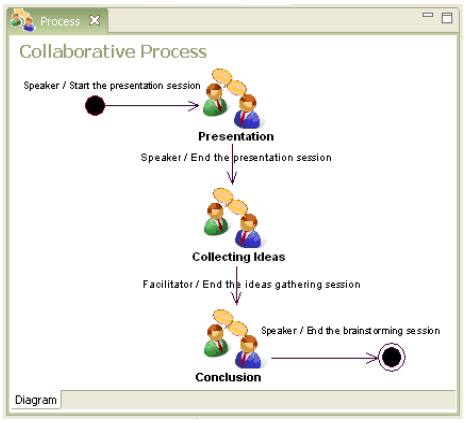

Figure 9. Concrete syntax of the metaclasses from the Protocols package.

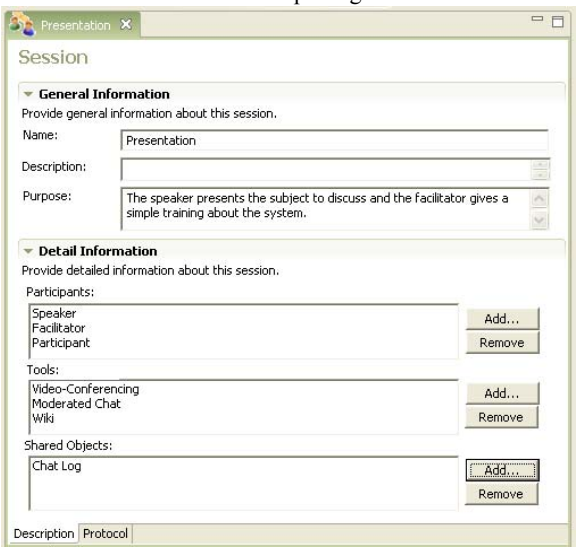

Figure 10. Concrete syntax of the Session metaclass

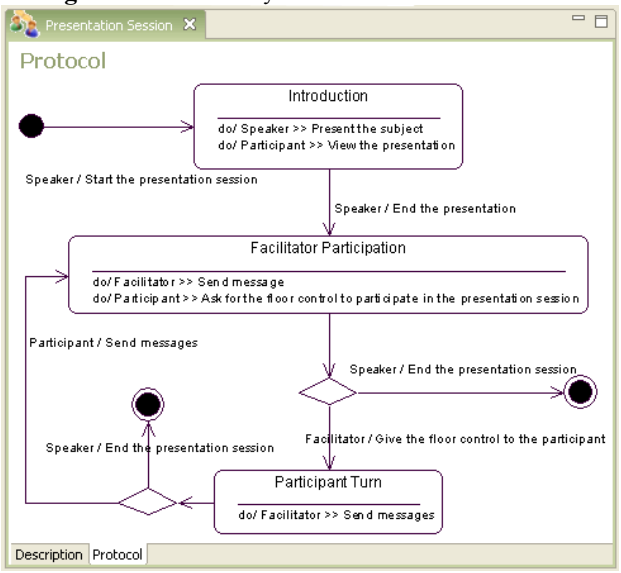

Figure 11. Concrete syntax of the metaclasses from the Protocols package (detailed view)

6. Conclusions

The construction of collaborative software is a very complex task. In the last years a number of toolkits and frameworks [15] [16] - that implement many common features of this kind of systems - have become available, facilitating its development. But, in order to improve the quality standards, support reuse, manage complexity and smooth the process of evolution and adaptation to new technologies, those frameworks are not enough. Such drawback is mainly due to the fact that those tools typically provide implementation-related usage examples that are focused on programming rather than modeling. Sometimes users' guides include modeling suggestions and ideas, but they are written using the language and concepts of the framework or toolkit. Apart from those platform specific tools, it would be desirable for collaborative software developers to count with abstract and platform independent design techniques and tools.

In this context and mainly due to the fact that the MDD paradigm has shifted the focus of software development from code-centric to model-based, the application of the MDD's principles to the development of collaborative systems is a promising approach to overcome the cited negative aspects.

Towards the application of MDD to collaborative systems development, in this paper we have presented CSSL, a domain specific language tailored to precisely match this particular domain's semantics. The CSSL constructs allow developers to build an abstract conceptualization of the domain in a precise but concise, friendly and simple way. Besides, the models written with CSSL are independent from any framework and toolkit. The language was defined as an extension of the UML by using the mechanism of metamodeling and it is supported by open source tools that we implemented on top of the Eclipse Platform [17]. Next step in this work will be to define a set of transformation from CSSL models to multiple collaborative systems frameworks making use of MDD technology and tools, such as QVT [18] and MofScript [19].

CSSL offers an improvement to the existing proposal by defining more complete, modular and structured language constructs for representing the groupware concepts. This language was defined and implemented using standard artifacts of MDD (i.e., a MOF metamodel and Eclipse) which favors its adoption in MDD development processes.

References

[1] Ellis, C.A., Gibbs, S.J., Rein, G.L., Groupware: some issues and experiences, in: Communications of the ACM, 34(1) (1991).

[2] Grudin, Jonathan. "Computer-Supported Cooperative Work: History and Focus". *Computer* 27 (5): 19-26. IEEE. ISSN: 0018-9162. (1994)

[3] Bran Selic. The Pragmatics of Model-Driven Development. IEEE Software, 20(5), 19-25 (2003)

[4] Stahl, M Voelter. Model Driven Software Development. John Wiley, ISBN 0470025700. (2006)

[5] Object Management Group, MDA Guide, v1.0.1 (2003)

[6] UML 2.0. The Unified Modeling Language Superstructure version 2.0 – OMG Final Adopted Specification,/2005-07-04. http://www.omg.org. (2005).

[7] Mandel, L. and Koch, N. and Maier, C. Extending UML to Model Hypermedia and Distributed Systems. From http:// projekte.fast.de/Projekte/forsoft/intoohdm/

[8] Rubart, Jessica and Dawabi, Peter. Shared data modeling with UML-G. International Journal of Computer Applications in Technology, Volume 19, Nos. 3/4, 2004.

[9] Dong, Ying and Li, Mingshu and Wang, Qing. A UML Extension of Distributed System. Proceedings of the Fist IEEE International Conference on Machine Learning and Cybernetics, Beijing, 4-5 November 2002.

[10] Bauer, B., Muller, J.P., and Odell, J. `Agent UML: a formalism for specifying multiagent interaction', Agent-Oriented Software Engineering, Springer-Verlag (2001)

[11] Pinheiro da Silva, P. and Paton, N.W. UMLi: the unified modeling language for interactive applications. Proceedings of the UML International Conference 2000, LNCS, Vol. 1939, pp. 117-132. (2000)

[12] Meta Object Facility (MOF) 2.0 Core Specification. OMG - (2005).

[13] Object Constraint Language OCL 2.0. OMG Final Adopted Specification. Document ptc/03-10-14. (2003).

[14] UML2. The Eclipse Model Development Tools (MDT) web site, http://www.eclipse.org/modeling/mdt/. It was accessed in May 2008.

[15] Tietze, D.A. (2001) `A framework for developing compo-nent-based co-operative applications', GMD Research Series No. 7/2001, ISBN: 3-88457-390-X.

[16] Guicking, A., Tandler, P. and Avgeriou P.Agilo: A Highly Flexible Groupware Framework. In Book Groupware: Design, Implementation, and Use. LNCS, Springer, Vol. 3706 (2005)

[17] Eclipse - an open development platform – http://www.eclipse.org

[18] Query/View/Transformations (QVT) - OMG Adopted Specification. March 2005. http://www. omg.org.

[19] MofScript. - www.eclipse.org/gmt/mofscrip

[20] Kelly, Steven and Tolvanen, Juha-Pekka.Domain Specific Modeling. Enabling Full Code Generation. John Wiley & Sons. (2008)