Collaboration Contexts, Services, Events and Actions: Four Steps Closer to Adaptive Collaboration Support in IMS LD

Florian König, Alexandros Paramythis
Institute for Information Processing and Microprocessor Technology (FIM)
Johannes Kepler University
Altenbergerstraße 69, A-4040 Linz, Austria
koenig@fim.uni-linz.ac.at, alpar@fim.uni-linz.ac.at

Abstract— **The IMS Learning Design specification, a widely known language for modelling collaboration scripts, has been criticized for a number of shortcomings and general lack of support for comprehensive adaptation features. We propose concrete extensions to the specification, aiming to alleviate deficiencies by specifically addressing (group) collaboration contexts, flexible service specification, fine-grained event handling and a wide range of adaptive interventions to support learners.**

Keywords—collaborative learning; adaptive support; learning design; IMS LD; extension; collaboration context; environment; services; event mechanism; adaptation actions;

I. INTRODUCTION

According to learning theorists, learning is a social activity, which can benefit from a collaborative setting [1]. In collaborative learning situations, certain interactions among learners can trigger learning mechanisms. Problem solving in a group with distributed knowledge, cognitive and real conflict between members, mutual questioning and discussion leading to collective sense-making and ultimately shared agreement and understanding can result in improved learning outcomes [2]. There is, however, no guarantee that beneficial interactions occur, even less so in distance learning, where lack of or limited 'real-world' contact amongst learners can negatively influence social- and group learning- patterns [3]. Especially in collaborative e-learning, support is needed to increase the probability that the desired interactions occur, because teams may not have worked together before, are usually formed for a comparatively short time, and individual learning goals are predominant. One way to support learners is by scaffolding their interaction in order to get group work going, mitigate disorientation and reduce cognitive load [4]. Possible support strategies range from anticipatively structuring the collaborative process (to favour the emergence of productive interactions), to retroactively regulating interactions, like tutors do [5]. One strategy along this continuum is to provide a detailed specification of the 'collaboration' contract in a scenario, which sets up systematic differences among learners in order to trigger contentious interactions, or to make rich interactions for exchanging complementary knowledge necessary [2]. Such a scenario is commonly called a 'collaboration script' [5].

In computer-supported collaborative learning (CSCL), so called CSCL scripts [6] exist as computational representations of collaboration scripts. They contain instructions specifying how members of a group should interact and collaborate to attain a task [7]. Static scripts, however, represent idealized scenarios and there is a danger of over-scripting [5], where fruitful collaboration is impeded by an overly coercive script. To adjust scripts at run-time, reduce its scaffolds over time (*fading*) and target the real needs of (groups of) learners, adaptive collaboration scripting has been proposed [8],[9] and found to be an effective method [10].

The most widely known formalization for CSCL scripts is the IMS Learning Design (IMS LD) specification [11], a learning process modelling language intended to formally describe designs of teaching-learning processes for a wide range of pedagogical approaches [12]. Still, IMS LD has been criticized both for insufficient expressiveness in aspects of the collaborative learning process [6], and for the absence of constructs that are vital in supporting adaptivity [13]: The language provides insufficient support to model group-based, synchronous collaborative learning activities and collaboration contexts [6]. The definition of services for providing, amongst others, collaboration facilities has been found to be rather inflexible [14] with support for only a limited number of service types, poor modelling of their characteristics, lack of insight into them once they have been instantiated and no means to manipulate them via adaptation interventions. IMS LD is also missing an event model and has only very few functions to modify a collaboration process at run-time [13].

This paper builds upon that criticism, as well as on extensions of the specification, that have already been developed (section II) or proposed in the literature (section III). We present a comprehensive set of modifications and additions to IMS LD, aiming to address many of its shortcomings and lay the foundations for improved support for adaptivity in CSCL scripts through: environments as representations of collaboration contexts (section IV); a flexible service specification (section V); a fine-grained event model (section VI); and, adaptation actions for modifying a scripted scenario at run-time (section VII). Section VIII summarizes the goals that can be achieved through the proposed amendments, and provides an outlook on issues to be tackled in future work.

II. IMS LD INFORMATION MODEL EXTENSIONS

In order to address the general shortcomings of IMS LD and those precluding its effective use in supporting collaborative and adaptive learning (see section I), certain extensions to both its information model and its run-time behaviour have already been developed and described in detail [15]. As

these extensions form the basis for the work presented here, their most important aspects will be summarised in the remainder of this section.

A simplified model of the run-time objects and their inter-relations can be seen in Fig. 1. Elements added to or changed from IMS LD are shown in gray. The specification has been extended as follows: Groups can be explicitly modelled (*group*), either statically or via specifying constraints for run-time creation of dynamic groups. Group members can share group-specific properties and an environment that acts as a common group workspace. A choice of different policies gives control over the grouping of participants for populating groups at run-time (*grouping*). For roles, properties and shared workspaces can be specified as well. Casting participants into roles is now possible at run-time via policies similar to those for grouping (*casting*). Grouping and casting operations can be sequenced by means of two new activities: *grouping-activity* and *casting-activity*.

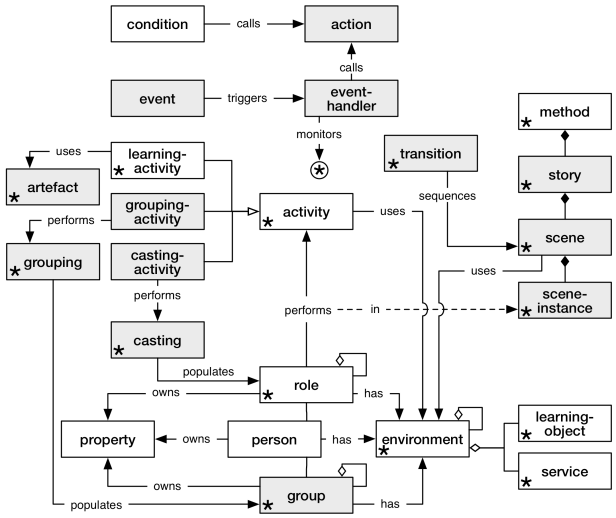

Figure 1. Object model (partial). New objects in gray.

To avoid having to rely on the semantics of properties for representing results of individual and group work, artefacts can be modelled explicitly, and their flow between learning activities is expressed by referencing them as input-, output- or transient- artefacts. Coupled with permissions, this defines how they progress in the script process and who can contribute.

The new sequencing model supports running multiple activity instances according to the social plane (one per class, group, individual), employs *transition* and *scene* elements for arbitrary sequencing of activities and uses workflow semantics for synchronizing and splitting the flow of action. A script can have multiple *story* containers (corresponds to IMS LD element *play*), which can each have multiple *scene* elements (correspond to combination of *act* and *role-part*). Scenes are connected by (conditional) transitions to form a directed graph, which leads from the start scene to the end of the story. Scenes tie actors (members of one or more *role* objects) to activities and specify the granularity of instantiation: one instance for all actors, one instance per specified group or one instance per actor. With this mechanism, collaboration contexts for groups can be created at the process level.

To implement advanced adaptation features in a script, knowledge of its run-time state is required. The proposed extensions to IMS LD feature a run-time model with fine-grained access to all elements of the script (see [15]). This model also contains pure run-time data, such as information on participants. Access to the run-time model is possible through new expressions, which made some operators obsolete (e.g., *users-in-role*) and new ones necessary (e.g., *run-time-value*). More flexible and readable definitions of adaptation rules are made possible through an event handling mechanism with event-condition-action (ECA) semantics (*event-handler*). These *event* objects and the range of new *action* objects for run-time scenario adaptation will, amongst other extensions, be described in detail in this paper.

III. Related Work

A number of extensions have been proposed in recent years to address shortcomings of IMS LD (see section I) and provide missing functionality. IMS LD lacks explicit collaboration contexts other than collaboration via a so-called "conference service". Miao et al. [6] have proposed running multiple instances of activities, if required by the respective social plane (one per role/group/person) to allow, for example, groups to work in parallel on the same problem. Their work, however, describes only how this changes the process model and not how this is reflected in the environments, which make up the other part of collaboration contexts (see section IV). In a similar effort, Hernández-Leo et al. [16] have proposed the "groupservice" for specifying collaboration spaces with communication facilities, interaction constraints, floor control policies, different interaction paradigms and awareness features. This approach provides many features missing in IMS LD, but suffers from the same limitation as the original specification: the actual collaboration happens outside the script's specification and control [17].

The problem of flexible service specification has been addressed by Valentin et al. [18] in their *Generic Service Integration* approach. It allows automatic service selection according to keywords specifying the expected service behaviour, author-defined alternative services, and explicit permission specification. Permissions are defined for different actions (read/write/admin) and object types (contribution, context), which, considering the wide range of possible services, is rather inflexible. Collaboration contexts are created by specifying the appropriate multiplicity (one-per-group, one-per-user, one-for-all, one-per-role) of a service but are not reflected in the process model. The possibility to specify an action that should be performed after a service has been closed hints at rudimentary event handling. A complete event model has not been included in any extension proposal yet.

With regard to adaptation, Miao et al. [6] propose to extend IMS LD with operations concerning activities, artefacts, roles, groups, persons, transitions, environments and relations between them. For creating re-usable fragments of "code" they propose to support action and expression declaration. Other suggestions for adaptation actions include

invoking system facilities, manipulating the system state (e.g., initiating communication sessions) [19], varying the group size, recommending or assigning (changes in) roles for participants, modifying the activity structure (e.g., adding / removing / reordering tasks) and determining the availability of elements (activities, services, artefacts) [13].

IV. ENVIRONMENTS

In IMS LD, activities are performed in the context of environments, containing learning objects (external resources), services (programmatic resources like a conference service) and other environments. In the original specification, each environment in the script source translates to one instance of that environment at run-time. Every participant in an activity is placed in that same environment specified for the activity. As IMS LD does not aim to support (group) collaboration, it does not need to provide different environment instances for different groups of participants in the same activity.

In the extended learning design model, similar to the original one, environments can be declared for learning activities and scenes, to specify a work context for when activities are instantiated. In order to support collaboration contexts at various levels, the number of scene instances can be configured in the extended specification: one for all actors, one per participating group, or one per person. Therefore, an environment defined for the learning activity and/or scene merely acts as specification for the individual environment instances created for each scene instance. If an environment is specified in both activity and scene, the components of the two are merged in the environment instance.

The instantiation mechanism is also used in the context of groups and roles. Independent of activities and scenes, each group and role can have an environment as its own "workspace". For groups it can, for example, contain communication and awareness facilities. Members of roles might benefit from a description of the role's obligations and tasks. These environments can be specified for each group and role individually. Alternatively, environments declared in groupings or castings act as specifications for the environment instances to create for each assembled group or casted role.

In the real world, as an instructional scenario proceeds, items such as learning material, assignment results and feedback accumulate. This can simultaneously happen at three main levels: for each single participant, in groups, and in the class. In the extended IMS LD model, this behaviour is mimicked: The contents of environments encountered during the run of a learning design script are accumulated at each level, on which they were introduced. For a scene, of which only one instance is run (i.e., class context), the contents of the scene and activity environments are added to the workspace of each role that is specified as an actor in that scene. With one instance per group, the environment is added to each group's workspace, and when running one instance per person, the environment is added to each personal workspace.

Both the merging of scene/activity environments and the aforementioned accumulation mechanism require the definition of precedence rules. Like in the original IMS LD specification, parts of an environment can be hidden or made visible via the static description or by adaptation rules. Precedence of such properties is defined by a partial ordering of the relevant objects, shown in Fig. 2: Settings for environments of objects further to the right have precedence over settings made in the context of objects to the left. Environment settings of a learning-activity are overridden by those in a scene, which in turn are overridden by those for specific instances of a scene. Role and group environment instance settings take precedence over settings in the environments of their casting or grouping policies. Precedence for roles, groups and persons follows the specificity of these objects.

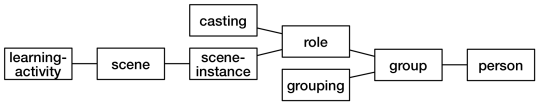

Figure 2. Environment precedence and propagation relation

Changes made to a more general environment specification can propagate to specific environment instances, if they have not been overridden there. Fig. 2 shows how changes are propagated between objects from left to right: Changes to a scene's environment specification, for example, propagate to instances of that scene, and possibly to role, group or personal workspaces. For instance, changing the visibility of an item in a grouping's environment affects the workspaces of all groups originating from the grouping. This mechanism allows for managing many environment instances at once.

As discussed before, in the script source only templates for environment instances are specified. They can be referenced via an identifier in the static description of the learning design. At run-time, multiple environment instances can be created from each template. To access these instances, the *get-instance* operator is introduced. It requires a static reference to an environment and a run-time reference to one of the objects in Fig. 2, and returns the instance created from the template as it is used for the specific object, allowing adaptation actions to target individual instances.

V. SERVICES

As mentioned in section I, the definition of services in the IMS LD specification has been criticized as being rather inflexible. To support the plethora of services existing in today's learning and collaboration platforms, a generalizable approach to specifying the requirements for the requested service is needed. Additionally, the wide range of possible applications coupled with a diverse set of functions provided by the underlying tools mandate a fine-grained permission model. In the extension to the IMS LD model, the service definition specifies a service type, optional parameters, permissions and (optionally) references to definitions of other services that could serve both as alternatives if instantiating the original service fails, or as a hint for possible adaptations.

A. Service Type Specification

There are four ways of specifying the choice of service by its type: First, a service registered with the run-time engine can be referenced by a descriptor like *service:chat*. Certain standard services that can be found in most learning platforms have been defined under the general *service* prefix (*service:wiki*, *service:chat*, …) The engine is expected to

support these services directly and map them to appropriate implementations. When a script author wants to use services of a certain platform (e.g., because they provide required functionality), platform-specific services can be specified. They must be registered with the run-time engine and can be referred to by a descriptor with prefix and service name chosen by the administrator. By convention, the name of the platform should be used as prefix (e.g., *sakai*, *blackboard*) and a general tool name (e.g., *forum*, *wiki*) or the name of a specific product (e.g., *wimba*, *elluminate*) as the service name. If multiple services provide the desired functionality, a list of these can be specified in the script. Each will be looked up in the engine's registry and the first one matching will be chosen. This allows for first requesting very specific services and falling back to more general ones later.

The second possibility of specifying the requested service type is by setting constraints and letting the run-time engine find a suitable service automatically. Constraints are either hard (must not be violated) or soft (may be violated, but the constraint solver still tries to avoid that). They can be prioritized by giving them a weight in the range of 0 to 1. If no weight is given, a default of 0.5 is used. The constraints can be broadly classified into the following categories:

- *function*: possible values are *communication*, *awareness*, *coordination*, *consensus*, *cooperation* and *collaboration*. These categories are not mutually exclusive. For instance, a forum is primarily a communication service, yet can still provide support for other tasks. The degree of support for each function must be set when registering a service with the engine.
- *interaction*: *synchronicity* (synchronous, asynchronous), *interaction mode* (implicit via mediating artefact, explicit direct interaction) [20] and *multiplicity* (1-to-1, 1-to-some, 1-to-many, many-to-many).
- *action coordination*: required modes of coordination between actors. Specific labels referring to certain strategies will need to be specified. At the moment, possible constraints are *free* (users interact freely), *moderated* (contributions are moderated) and various *floor control* policies (see section VII.C).
- *awareness*: required indicators for *social awareness* (peer presence, motivational state, attitude, individual and group communication activities, …) [21], *group-structural awareness* (peers' roles, responsibilities, positions, status, …) [22], *action awareness* (interaction with shared resources, location and focus of current activity, …) and *activity awareness* (shared plans, rationale, task dependencies, …) [21].
- *modality* of interaction (communication) as either *text*, *graphics-2d*, *graphics-3d*, *audio* or *video*.
- *artefacts*: type of item supported for *viewing* or *creation* specified on three levels: atomic (simple text, rich text, hypertext, image, video, sound), compound (e.g., presentation, animation) or via MIME types.
- *shared environment*: features of the service's workspace like *degree of what-you-see-is-what-i-see* (relaxed, strict) [20] or *tele-pointer* support [23].
- *privacy*: features like the possibility for *data hiding* (controlling the visibility of contributions to peers) [20], availability of a *private workspace* in addition to the shared workspace [24], support for *anonymous participation* or completely *hidden participation*.
- *technical*: number of *supported users* (arbitrary, minimum, maximum) and *session persistence* (none, partial-history, full-history, results-only)

The third possible way of specifying the service type is by directly referencing the service via a URI. This can be a URL with the service being independent from the run-time engine, not managed by it and no back-flow of (event) data. Alternatively, engines could support control and event features via, e.g., web service- or REST-style interface(s). The fourth way to specify a service type makes human intervention possible. Using the three other service specification methods, a set of service choices can be prepared. At run-time, a human participant must choose one to be instantiated.

In IMS LD, all services have to be instantiated before the script is started. This early binding is still possible when the service has been specified via a registered descriptor or directly via a URI, and, under certain conditions, also for services defined by constraints. Late binding of services, one of the prerequisites for adaptation, is applied when constraints refer to run-time information (e.g., *supported users* constraint set to number of participating users), or when the service should be selected manually. In the later case, an adaptation rule might take over this task and use knowledge about the participants to tailor the service choice to their needs.

B. Service Functions

For specifying the permissions of roles in relation to a service (see section V.C), modelling the events that a service can generate (see section VI.B), and defining how adaptation actions can control it (see section VII.C), a model of the service's functionality is required. The run-time engine, where each supported service has to be registered, needs to know which functions a service provides to its users. In particular, the engine needs: a name for referring to the function; a way to effect function-specific permissions; instrumentation to be notified upon function use; and (optionally) an interface to call the function with any required parameters. To keep the integration and configuration effort at a reasonable level, and prevent over-specifying the interface to services, a small set of general functions has been created. The basic functionality of any service should be mapped to these baseline functions:

- *new*: create content item/contribution/utterance
- *read*: perceive content/contribution/discussion
- *revise*: change existing content
- *delete*: delete content/contribution
- *moderate*: publish item on behalf of somebody else
- *communicate*: reply, comment, out-of-band communication (e.g., text chat in whiteboard)
- *create/remove section*: manage sub-division of (interaction) space provided by service
- *enter/leave section*: access to sub-workspace
- *archive*: persist/export the content/results

Additional functions exist for synchronous services:

- *start/end session*: manage time-limited provision of (interaction) space in service
- *enter/leave session*: access to (interaction) space

If the service supports floor control, then the following functions can be used to model the primitives for expressing various policies [20]: *request-floor*, *release-floor*, *assign-floor*, *revoke-floor*.

The functionality of tools commonly found in LMSs can be mapped to the above functions. For example in a wiki, one can create (*new*), view (*read*), edit (*revise*) or delete pages (*delete*), moderate contributions (*moderate*), comment on pages (*communicate*), manage/access topical areas (*section*), and export content (*archive*). In an A/V conferencing tool, one can make an utterance (*new*), manage speakers (*moderate*), converse in a text chat (*communicate*), manage/access meetings (*session*), manage/enter private rooms (*section*) and export a session (*archive*). Other functions of a service that cannot be readily mapped to general functions (e.g., *vote* in a polls tool) must be separately registered with the run-time engine. Other than that, there is no difference in usage between a general and a service-specific function.

C. Permission Specification

Instead of relying on fixed categories such as *observer*, *participant*, *moderator* and *conference-manager*, as defined in IMS LD, an approach for fine-grained control over which role may perform which function is proposed. Permissions are defined for specific functionality of a service, referred to by an identifier: *prefix.service-name.function[.own|.any]*. Prefix and service name correspond to the values used to specify the type of service to instantiate (see section V.A). The function name (see section V.B) defines the context of the permission, an optional *own* or *any* can differentiate between the right to, for instance, revise one's own blog posts or modify anyone's posts. For each function, a positive (*allow*) or a negative (*deny*) permission can be created.

Permissions are assigned to one or more roles in the script. They can be specified by either defining a set of them "from scratch" or by using (and possibly extending or overriding) a service-specific profile that provides reasonable default permissions. Each service can have any number of profiles, identified by a name (e.g., *reader*, *tutor*) and comprised of defaults that this profile represents. For the general services under the *service* prefix, appropriate profiles are defined in the specification. For other services, profiles have to be defined via configuration mechanisms of the run-time engine. When using a service in a script, instead of having to create a complete set of permissions, a profile can be referenced and extended/overridden with custom permissions.

Due to the way that the service type can be specified (see section V.A), it may not be known at design time which service will be instantiated. Functionality identifiers seem to be service-specific, but specifying permissions is still possible: One approach is to define permissions for the functions of all services that can potentially be instantiated. This works, for example, when using a list of registered service descriptors for requesting a service. In cases where the set of possible services is not known a priori, wildcards in the function name can be used for the prefix and the service name. The function identifier **.*.delete.any* would represent the permission to delete any content element in any service on any platform. With **.blog.delete.any* this can be constrained to just blog services, and with *general.*.delete.any* to any service in the set of general services described in the specification.

Permissions are inherited throughout the hierarchy of roles. This allows for giving a limited set of permissions to general roles like *learner* or *staff* and providing extra permissions to more specific roles like *group-leader* or *administrator*. To resolve conflicts arising when permissions are both inherited and defined in a role, there are two resolution rules: *deny* permissions have precedence over *allow* permissions; and permissions with more specific identifiers override those with more wildcards or a wildcard at more general positions.

VI. Event Model

Fundamental to adaptation rules are the criteria that need to be fulfilled to trigger the rule's action(s). In IMS LD, rules are expressed as conditions, which are triggered by Boolean expressions referring to properties. Some problems exist with this approach: First, all state changes where an action should be triggered need to be recorded in properties, with a property required for each dimension of the state model. In complex scripts, this requires a large number of properties, although most of the information would be contained in the run-time state of the script engine anyway. Second, the properties contain no semantic information about their purpose, except in their name, which cannot be used for automatic inferences. As a consequence, conditions lack semantics as well. Third, properties and conditions form an unstructured base of state and adaptation information, which makes them difficult to understand and use, and might create performance problems when many conditions have to be evaluated every time a property's value changes. Fourth, values of properties can only be changed upon completion of certain script components (activities, *act*, *play*, *method*) completes, or users change them manually. This represents a big restriction on the types of state changes that can be detected at all.

To address these shortcomings, extensions to IMS LD have been proposed, replacing the *on-completion* hook (in activities, acts, plays and methods) with a general event-condition-action (ECA) handler concept [15]. Each event handler can process events specific to the context in which it is specified, can have a filter expression to specify the exact conditions when it triggers actions, and has a list of actions (see section VII) to perform once triggered. This approach provides the following benefits over the original model: state changes do not need to be modelled in properties but can be "detected" by appropriate event handlers; the set of possible events has defined semantics, which allows reasoning about them by components other than the event handlers themselves; event handlers are defined at the element which they monitor, which models their semantic relation and improves readability; finally, event handlers can be defined for most elements of a learning design script, which allows for fine-grained reactions to state changes (see sub-section VI.A).

A. Script Events

Script events originate from the objects shown in Fig 1 with an asterisk (*), which are those elements for which event handlers can be defined. Possible script events are:

- *activity, scene-instance, scene, story*: started, paused, resumed, cancelled, completed, completed-repeatable, participant-joined/-left
- *transition*: fired
- *learning-activity*: artefact-delivered
- *artefact*: created, read, updated, (un)locked, deleted
- *role*: member-added/-removed, subrole-added/-removed, property-added/-removed
- *group*: member-added/-removed, subgroup-added/-removed, property-added/-removed
- *casting*: permitted-methods-shown, started, casted, casted-with-exceptions, review-started, rejected, finished, finished-with-exceptions
- *grouping*: permitted-methods-shown, started, grouped, grouped-with-exceptions, review-started, rejected, finished, finished-with-exceptions
- *learning-object*: reference-changed
- *service*: (un)deployed, participant-joined/-left
- *environment*: learning-object-added/-removed, service-added/-removed

These events have been chosen to cover practically all relevant changes in the state of the individual objects.

IMS LD provides a way to complete activities (and the elements for sequencing them) after a time limit. This approach has been generalized with so-called timers. For each element that can produce events, timers can be specified to trigger a *timer* event, which can in turn trigger arbitrary actions via an event handler. Optionally, timers can be defined to trigger recurrently with a certain frequency, either until a certain time or for a certain number of triggerings. Timers can be used, for example, to set the state of an activity after some time, like in IMS LD; to regularly give feedback to participants; to rotate roles in a group every week; etc.

B. Service Events

Events returned from a service are closely related to its functionality. Registered general and service-specific functions (see section V.B) define, which events exist for a service. There is a wide range of possible data that can be returned from a service by an event. To prevent over-specification, only a reference to the service, a timestamp, the participant causing the event, and (if applicable) identifiers for the section and the affected object in the service's workspace are mandated. Modelling mechanisms requiring more detailed information should be kept outside the script source, where they can have custom interfaces to services, receive a wide range of data and employ arbitrarily complex algorithms to create user and group models. Via a connection with the run-time engine, these mechanisms could populate the run-time model of a script with their models.

VII. Adaptation Actions

The original IMS LD specification provides only a limited number of actions: showing/hiding objects, changing the value of properties and giving feedback. Comprehensive support for adaptive interventions, however, requires a wide range of additional actions in order to effect meaningful changes on the execution of a learning design. The necessary actions can be broadly categorized into the following classes:

- *object adaptations*: creating, modifying and destroying objects (group, scene, artefact, …)
- *relation adaptations*: modifying attributes of objects or processes in relation to other objects or processes (membership, ownership, permissions, visibility, …)
- *control flow adaptations*: starting, stopping, modifying (e.g., new branches) the script process
- *environment adaptations*: managing services and performing human-like actions in the environment
- *adaptation control*: managing the adaptations themselves (conditions, event handlers, actions, …)

These actions cover the life cycle of objects, relations between them, control of active processes, adaptations of external services and basic building blocks for meta-adaptivity. Three of the classes will now be described in more detail.

A. Object Adaptations

Adapting objects as such deals with managing their life cycle and adjusting those attributes and parameters at run-time that do not represent relations to other objects. This focuses on the properties of objects that are self-contained, yet may still show effects outside the object itself or may be constrained by external circumstances.

Any object except *learning-design* and *person* objects can be created at run-time by adaptation actions. Mandatory attributes and relations according to the specification ([11], [15]) must be provided as parameters to the creation action; optional attributes may be provided as well. When calling the *create-role* action, for example, title and type are mandatory, and references to detailed information on the role, as well as to a role environment, are optional. When a role environment is specified, this also has an effect outside the target of the adaptation action: an environment instance is automatically created to serve as shared environment for role members.

Most attributes of objects can be modified at run-time, yet some constraints exist: first, multiplicities of attributes must be respected (e.g., mandatory ones must not be "removed"); second, the new value must be in the range of valid values; third, some attributes can only be changed when the object is in a certain state: for example, changing which groups should be assembled by a grouping only works before the grouping has started. If an action violates such a constraint, an exception is generated. Script designers can define mechanisms to detect exceptions and react accordingly. Due to lack of space, the exact workings of exceptions in the context of a learning design script can not covered in this paper.

Destroying objects is theoretically possible, yet in practice they are often tightly interconnected (see Fig. 1), requiring prior "detachment". Constraints may prevent objects from being destroyed, for example when they are part of an ongoing process (e.g., running *scene-instance*). Checking all constraints and severing all connections to other objects would require a lot more operations than a single action, and may not be feasible to specify at design-time without detailed knowledge of the specific situation. One approach of removing objects from the immediate context of a run, but leaving them in place to retain referential integrity, can work as follows: Alternatives to the object can be prepared at design-time or at run-time. For example, an alternative role or a

path in the sequencing of the story can be provided. The alternatives need to be connected to objects just as the original ones, so in our example the new role must be added as actor to all the places where the original role was used, and the alternative path may be connected to the main path via conditional transitions. The next step is to transfer the "active" elements to the alternative object or make sure that this is done during the rest of the script run. For roles, this means moving the members to the new role. The new sequencing path can be activated by making sure that conditional transitions from preceding scenes direct the control flow along it, instead of along the original path. What remains is replacing any references to the original object (e.g., in conditions) with ones to the new one. Ideally, the object would be ready for destruction afterwards, but this step is not always necessary.

B. Relation Adaptations

Adaptations to the relations between objects deal with all the attributes of objects that link them to other objects. The relations that can be adapted can be categorized as follows:

- *containment*: method↔story↔scene↔scene-instance, environment↔learning-object/service/environment, property-group↔property/property-group
- *hierarchy*: role↔sub-role, group↔sub-group
- *membership*: role↔person, group↔person
- *task*: scene↔activity, scene-instance↔activity
- *actor*: scene↔role, scene-instance↔role/group/person
- *environment*: activity↔environment, scene↔…, scene-instance, person, role, casting, group, grouping
- *sequencing*: scene↔transition↔scene
- *ownership*: person↔property, grouping↔property, group↔property, role↔property, casting↔property
- *permissions*: service↔role, artefact↔role
- *artefacts*: learning-activity↔artefact
- *visibility*: learning-activity↔artefact, environment↔learning-object/service

Relations are managed by a *set* operation and, depending on the multiplicity, by an *add* and a *remove* operation. Role members, for example, can be defined by *set-members*, or changed, one by one, with *add-member* and *remove-member*.

C. Environment adaptations

In IMS LD, services are "black boxes" that can be neither monitored nor manipulated. However, a lot of the interaction happens in the environment, specifically in services. Therefore, adaptively supporting users of services is a way to advance from collaboration establishment to active collaboration process support [13]. It can be achieved by tuning service functionality and operations to its users, and by effecting actions that would be normally performed by a human.

Adaptation can first happen when the service is automatically selected, taking into account needs and characteristics of participants and groups. While a service is running, the permissions of roles can be adapted, for example to give a group leader moderation rights. Permissions can also be used to selectively enable specific functionality. If, for instance, replies can be disallowed in a forum service, this can be used to emulate a brainstorming tool. As customary, participants can then only create new postings with ideas but cannot (directly) comment on ideas. Later, reply permissions may be granted, to allow a discussion of the collected ideas.

Coordination mechanisms like floor control can influence the interaction in communication and collaboration tools. The floor control policy of a service may be changed in order to make interaction more free-form and minimize the scaffolding, or, conversely, to switch to a more coercive policy. Examples of policies are [20]: a moderator assigns the floor to a contributor who returns it after use; the current floor holder passes the floor to a contributor who requested it; all floor requests are queued and upon floor release the first one in the queue receives it. The first two policies (and possibly others as well) can be specified with appropriate permissions for the *request-floor*, *release-floor*, *assign-floor*, *revoke-floor* functions (see section V.B). Policies that have a state (e.g., the queue in the third one) require explicit support from the service. Policies must be specified in the service interface and may be enacted by an action like *set-floor-control-policy*. Alternatively, directly calling a service's *assign-floor* or *revoke-floor* actions allows for direct intervention.

In services for synchronous interaction, means to start and end sessions and invite participants would allow to set up ad-hoc communication/collaboration contexts. For example, if it has been detected that a group needs help, a communication session (e.g., chat) could be started and its members and an instructor could be automatically invited.

Performing adaptive interventions in the environment in a manner akin to that of human operators (e.g., a tutor) can be used to provide feedback and content scaffolding. This requires adaptation actions that directly put content into running services. A full specification of all functions for managing content in a service would result in over-specification, yet some functions of certain services can be called from inside a script. This applies especially to the general functions *new* (create content item) and *create-section* (create subdivisions like conference rooms, forum topics). Actions calling these functions with appropriate parameters could express interventions like: give feedback or advice in a text chat, for example in the form of advice phrases like "Invite others to participate" or "Reflect with teammates about …" [25]; provide structures like forum topics or folders in a file space to help participants organize their collaboration; provide content scaffolding by creating new (example) items in a glossary or (template) pages in a wiki; etc.

VIII. Summary and Outlook

In this paper, we have presented extensions to IMS LD that aim to allow for better expressing collaboration scripts and more comprehensive adaptation behaviour. For representing collaboration contexts, we have extended the environment model to support workspaces for whole classes, individual groups and single users. A flexible service model has been introduced, which allows fine-grained service selection and permission configuration. With an event model, a wide range of state changes in run-time objects and services can be detected and used to trigger actions. Finally, adaptation actions have been introduced for manipulating objects and their relations at run-time, and for effecting interventions in collaboration environments, specifically in services.

In order to ascertain the expressiveness and suitability of the extensions in practice, we are currently assessing them against a representative set of existing scripts and collaboration patterns. The results of this evaluation will guide further extension efforts. For future iterations of this information model, we will address the following aspects: an exception handling mechanism for dealing with unforeseen situations and for increasing execution robustness of scripts; actions for control flow adaptations and adaptation control as well as support for provisional adaptation decisions; a formalism for creating expressions and action declarations to allow re-using sets of actions and expressions across the script; and control structures (loop, branch) and transactional processing semantics for sequencing of adaptation actions.

Parallel to this, we have started to implement the specification in an executable model, with run-time components capable of running the script, and services, user interface and infrastructure provided by the Sakai e-learning platform [26]. This prototype will then be employed in real-world student-based evaluations, where we will seek to establish the impact of the newly enabled types of adaptive support on the collaborative learning process. After thus providing the foundation for adaptive collaboration scripting, we will turn our attention towards developing tools that allow authors to exploit the potential of both the specification and the engine.

ACKNOWLEDGMENTS

The work reported in this paper has been supported by the "Adaptive Support for Collaborative E-Learning" (ASCOLLA) project, financed by the Austrian Science Fund (FWF; project number P20260-N15).

REFERENCES

[1] J. Roschelle and S. Teasley, "The construction of shared knowledge in collaborative problem solving," *Computer-Supported Collaborative Learning*, C. O'Malley, Ed., Berlin: Springer, 1995, pp. 69–97.

[2] P. Dillenbourg, "What do you mean by collaborative learning?," *Collaborative-learning: Cognitive and Computational Approaches*, P. Dillenbourg, Ed., Oxford: Elsevier, 1999, pp. 1–19.

[3] A. Paramythis and J.R. Mühlbacher, "Towards New Approaches in Adaptive Support for Collaborative e-Learning," *Proceedings of the 11th IASTED International Conference*, Crete, Greece: 2008, pp. 95–100.

[4] J. Zumbach, J. Schönemann, and P. Reimann, "Analyzing and supporting collaboration in cooperative computer-mediated communication," *Proceedings of th 2005 conference on Computer support for collaborative learning: learning 2005: the next 10 years!*, Taipei, Taiwan: International Society of the Learning Sciences, 2005, pp. 758–767.

[5] P. Dillenbourg, "Over-scripting CSCL: The risks of blending collaborative learning with instructional design," 2002.

[6] Y. Miao, K. Hoeksema, H.U. Hoppe, and A. Harrer, "CSCL Scripts: Modelling Features and Potential Use," *Proceedings of the 2005 Conference on Computer Support for Collaborative Learning - Learning 2005: The next 10 Years!*, Taipei, Taiwan: International Society of the Learning Sciences, 2005, pp. 423–432.

[7] A.M. O'Donnell and D.F. Dansereau, "Scripted Cooperation in Student Dyada: A Method for Analyzing and Enhancing Academic Learning and Performance," *Interaction in Cooperative Groups: The theoretical Anatomy of Group Learning*, R. Hertz-Lazarowitz and N. Miller, Eds., London: Cambridge University Press, 1992, pp. 120–141.

[8] N. Rummel, H. Spada, and S. Hauser, "Learning to collaborate while being scripted or by observing a model," *International Journal of Computer-Supported Collaborative Learning*, vol. 4, 2009, pp. 69–92.

[9] N. Rummel, A. Weinberger, C. Wecker, F. Fischer, A. Meier, E. Voyiatzaki, G. Kahrimanis, H. Spada, N. Avouris, E. Walker, K. Koedinger, C. Rosé, R. Kumar, G. Gweon, Y. Wang, and M. Joshi, "New challenges in CSCL: Towards adaptive script support," *Proceedings of the 8th International conference of the Learning Sciences*, Utrecht, Netherlands: International Society of the Learning Sciences, 2008, pp. 338–345.

[10] S. Demetriadis and A. Karakostas, "Adaptive Collaboration Scripting: A Conceptual Framework and a Design Case Study," *Complex, Intelligent and Software Intensive Systems, International Conference*, Los Alamitos, CA, USA: IEEE Computer Society, 2008, pp. 487–492.

[11] IMS Global Learning Consortium, Inc., "Learning Design Specification (Version 1.0 Final Specification)," 2003.

[12] R. Koper and B. Olivier, "Representing the Learning Design of Units of Learning," *Educational Technology & Society*, vol. 7, 2004, pp. 97–111.

[13] A. Paramythis, "Adaptive Support for Collaborative Learning with IMS Learning Design: Are We There Yet?," *Proceedings of the Adaptive Collaboration Support Workshop, held in conjunction with the 5th International Conference on Adaptive Hypermedia and Adaptive Web-Based Systems (AH'08)*, Hannover, Germany: 2008, pp. 17–29.

[14] M. Caeiro, L. Anido, and M. Llamas, "A Critical Analysis of IMS Learning Design," *Proceedings of CSCL 2003*, Bergen, Norway: 2003, pp. 363–367.

[15] F. König and A. Paramythis, "Towards Improved Support for Adaptive Collaboration Scripting in IMS LD," *Sustaining TEL: From Innovation to Learning and Practice – Proceedings of the 5th European Conference on Technology Enhanced Learning Sustaining (EC-TEL 2010)*, M. Wolpers, P.A. Kirschner, M. Scheffel, S. Lindstädt, and V. Dimitrova, Eds., Barcelona, Spain: Springer-Verlag, 2010, pp. 197–212 (in press).

[16] D. Hernández-Leo, J.I. Asensio-Pérez, and Y.A. Dimitriadis, "Computational Representation of Collaborative Learning Flow Patterns using IMS Learning Design," *Educational Technology & Society*, vol. 8, 2005, pp. 75–89.

[17] F. Jurado, M. Redondo, and M. Ortega, "Specifying Collaborative Tasks of a CSCL Environment with IMS-LD," *Cooperative Design, Visualization, and Engineering*, 2006, pp. 311–317.

[18] L. de la Fuente Valentin, Y. Miao, A. Pardo, and C. Delgado Kloos, "A Supporting Architecture for Generic Service Integration in IMS Learning Design," *Times of Convergence. Technologies Across Learning Contexts*, 2008, pp. 467–473.

[19] A. Paramythis and A. Cristea, "Towards Adaptation Languages for Adaptive Collaborative Learning Support," *Proceedings of the First International Workshop on Individual and Group Adaptation in Collaborative Learning Environments (WS12) held in conjunction with the 3rd European Conference on Technology Enhanced Learning (EC-TEL 2008). CEUR Workshop Proceedings, ISSN 1613-0073, online CEUR-WS.org/Vol-384/*, Maastricht, The Netherlands: 2008.

[20] W. Reinhard, J. Schweitzer, G. Völksen, and M. Weber, "CSCW Tools: Concepts and Architectures," *Computer*, vol. 27, 1994, pp. 28–36.

[21] J.M. Carroll, D.C. Neale, P.L. Isenhour, M.B. Rosson, and D.S. McCrickard, "Notification and awareness: synchronizing task-oriented collaborative activity," *International Journal of Human-Computer Studies*, vol. 58, May. 2003, pp. 605–632.

[22] C. Gutwin, S. Greenberg, and M. Roseman, "Workspace Awareness in Real-Time Distributed Groupware: Framework, Widgets, and Evaluation," *Proceedings of HCI on People and Computers XI*, Springer-Verlag, 1996, pp. 281–298.

[23] C.A. Ellis, S.J. Gibbs, and G. Rein, "Groupware: some issues and experiences," *Communications of the ACM*, vol. 34, 1991, pp. 39–58.

[24] C. Ellis and J. Wainer, "A conceptual model of groupware," *Proceedings of the 1994 ACM conference on Computer supported cooperative work*, Chapel Hill, North Carolina: ACM, 1994, pp. 79–88.

[25] M.D.L.A. Constantino-Gonzalez, D.D. Suthers, and J.G.E.D.L. Santos, "Coaching Web-based Collaborative Learning based on Problem Solution Differences and Participation," *International Journal of Artificial Intelligence in Education*, vol. 13, 2003, pp. 263–299.

[26] Sakai Foundation, *Sakai Project*, http://www.sakaiproject.org.