Deliverable N°8

Open Architecture

The LeActiveMath Consortium
December 2004

Main Authors:

Paul Libbrecht (DFKI), Editor,
Éric Andrès (DFKI), Oliver Lemon (Edin), Rafael Morales (UNN), Kaśka Porayska-Pomsta (Edin), Stefan Winterstein (DFKI), Carsten Ullrich (DFKI), Claus Zinn (Edin)

Project funded by the European Community under the Sixth Framework Programme for Research and Technological Development

Project ref.no.	IST-507826
Project title	LeActiveMath- Language-Enhanced, User Adaptive, Interactive eLearning for Mathematics

Deliverable status	Restricted
Contractual date of delivery	December 31st 2004 (Month 12)
Actual date of delivery	January 21st 2004
Deliverable title	Open Architecture
Type	Report
Status & version	1.0
Number of pages	85
WP contributing to the deliverable	WP3
WP/Task responsible	T3.1
Author(s)	Paul Libbrecht (DFKI), Editor, Éric Andrès (DFKI), Oliver Lemon (Edin), Rafael Morales (UNN), Kaśka Porayska-Pomsta (Edin), Stefan Winterstein (DFKI), Carsten Ullrich (DFKI), Claus Zinn (Edin)
EC Project Officer	Colin Stewart
Keywords	Open architecture

Contents

1 Executive Summary

1.1 About this document

This document represents the current state in the development of the architecture needed to achieve the very rich integration in the Le ActiveMath project.

Instead of packing a full-fledged architecture report which would have not been able to include all components, we decided to present an internal *components* document which presents the role of each components, its interfaces, and its use-cases in the current state of the research in the project. As a result, this document contains several parts which are left to be cleared-up and several open questions.

1.2 Architecture Goals

The description of work of the project [8], describes a truly opened architecture based on web-services to be essential to the project.

In the requirements analysis report of Le ActiveMath [9] an amount of *features* of the system are described. All of them have tried to be satisfied by this architecture.

The system expects to be used by at least a classroom of students and should offer a good responsiveness, it expects to be used over the web, with as few installations on the clients side as possible (claim 2.2).

This architecture thus presents the following ingredients: a description of the communication mechanisms between components, a list of components, and, for each component, a description of its role and conceptual functions and its ways of external communication.

This document, and its future versions, should thus serve as a reference guide to the planned Le ActiveMath system. We present, in this section, a short summary.

1.3 Environment

Le ActiveMath web server is expected to be cross-platform and be based on mature open source libraries. We intend to make the Le ActiveMath software deliverable: easily distributable, easily installable, and easily configurable, on most contemporary platforms.

1.4 Basic Server Infrastructure

Le ActiveMath is a web-server based on Java Servlet architecture [5]. It serves web-based presentation content which provides a user-interface to present learners content in a friendly and readable way, access to facilities related to this content, and access to functions supporting the learning experience.

Following user-interface best-practices, the Le ActiveMath server is organized according to a model-view-controller architecture [12]. It separates application logic (*controller*) from application data (*model*) and the presentation of that data (*view layer*). It is well established for Java web applications, where servlets act as controllers, Java data objects form the model, and views are usually implemented by a dedicated template language.

The controller classes are embedded in ActiveMath Front End and provide the application logic to support basic actions. They are responsible to provide to the views the necessary knowledge handlers' data.

Knowledge handlers form the bulk of the Le ActiveMath components. They handle, store, read, select, and updates knowledge, some specific to a learner.

Some knowledge handlers objects have a direct user-interface. For example, the tutorial component, the exercise system, or the dictionary are components that are directly attached with views and controllers. Most of the controllers, however, take advantage of several components and much of the components are addressed to as a *service* (for example, the learner-model).

By now, the framework for the open web-architecture of Le ActiveMath is specified, and it relies on a sservice interfaces and a generic event messaging mechanism. An advantage of this architecture is the generic integration of external as well as internal tools and the asynchronous exchange of information among all components of Le ActiveMath .

1.5 Components lifecycle

Components tightly integrated are created by the core Le ActiveMath. Following the java servlets practice, components are invoked on demand, only started at the first request. ActiveMath comes with an extensible configuration mechanism which offers subsetting so that a component can be configured with a *sub-tree* of the core configuration. Some components may also be defined to be services which are globally and permanently available. The configuration system allows the global availability of services responding to a given contract. This configuration mechanism has allowed the implementation of components that serve in such different settings as tiny-downloads restricted applet-environments or heavy-load servers.

Depending on their nature, components are integrated into Le ActiveMath using simple Java classes and methods or using remote, xml-rpc-based, communication. we have chosen, thus far, the xml-rpc [2] standard for web-services for its simplicity and very broad platform deployment, not being fully satisfied with any more modern web-service technology.

This remote and local *synchronous* communication provided by the function paradigm is, however, often inappropriate. It has thus been complemented by an event framework. The event paradigm provides the best programming paradigm for state update notifications and reactions thereof.

A good example of event propagation is started by the front-end, in the exercise-controller, indicating that the learner has performed an exercise with a given score. Listeners to such an event are, for example, the suggestion engine of the tutorial component (see 12) that can take this into account to suggest supporting explanations, the learner model which can take this into account to update its beliefs about the learner competencies, ...

1.6 Components Identified

Components have been separated in three classes depending *how close* they are to Le ActiveMath learning environment:

- The core-components form the essential part used in all situations. They include the basics configuration and communication services, the front end, and the user management.
- Tightly integrated components include all components of the project which live inside the server. These components take full advantage of available services. This includes:
  - the Presentation System
  - the Content Manager (including and wrapping the MBase and Metadata Query Mediator)
  - the Tutorial Component

  - the Dialogue Manager
  - the Learner Model
  - the Dictionary
  - the Assembling tool
  - the Exercise subsystem
  - the Concept Mapping tool

- Two other components are being integrated *loosely*: they have their own server-environment. The learners are being taken from one server to another for the duration of an *activity*. Communication is done in a collaborative fashion with limited interfaces and no centralized service. This includes:
  - the exercise repository
  - the Siette assessment tool

2 Graphical Overview

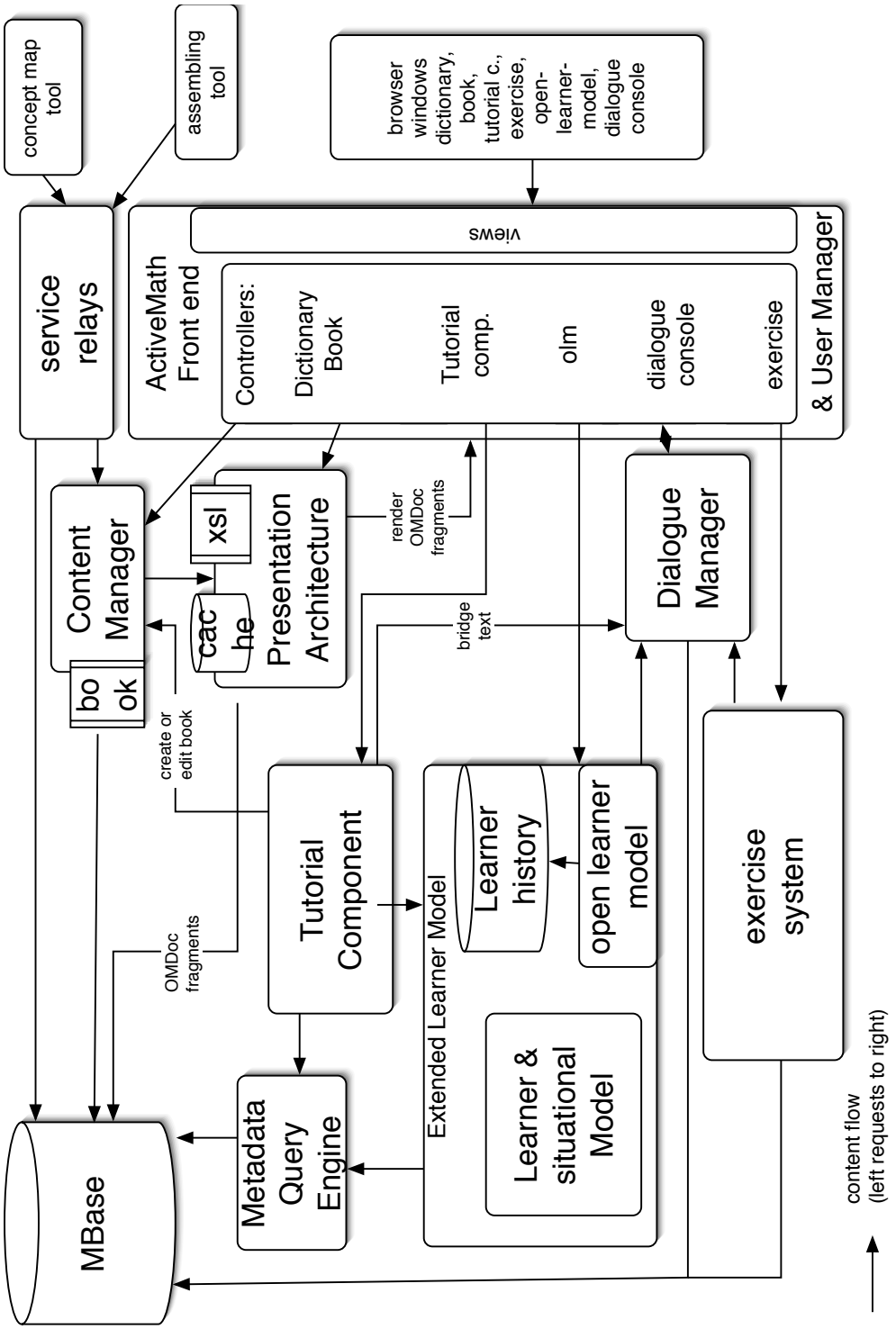

Figure 1: Le ActiveMath overall architecture.

3 Introduction

3.1 Le ActiveMath Components

The ActiveMath learning environment is a web-server infrastructure which supports active learning of mathematics. The Le ActiveMath architecture proposes a revised plan of the web-server to fullfill the requirements described as a result of the Le ActiveMath requirements analysis [9].

This architecture tries to decompose Le ActiveMath into components to define their role, interfaces, and expected information flows.

Each component in the Le ActiveMath architecture handles knowledge. Except for the ActiveMath Front End, each of them is, thus, viewed as a part of the model of the model-view-controller paradigm.

This document is organized as follows:

- Primary Components
- Tightly-integrated Components
- Loosely-Integrated Components

These chapters provide a description of the components information handling, its lifecyle and configuration, and document the interfaces as remote interfaces (using XML-RPC), as Java interfaces, and as events emitted. Each components provides a set of descriptive information-flows that expect to happen in typical situations between several components. They are here to illustrate the usage of the components' interfaces.

3.2 Communication Mechanisms

We briefly provide details on the chosen communication mechanisms.

For programmatic integration of components, ActiveMath has chosen the Java platform as a cross-platform deployment facility whose adoption is wide enough to provide a rich set of open-source third-party libraries.

For remote communication with components which don't live in the Java virtual machine, ActiveMath has chosen, thus far, the xml-rpc [2] standard for its simplicity and very broad platform deployment. Though a move towards SOAP is expected, the need for it has not yet been felt enough especially since no existing standard provide at least the following features:

- documentation mechanism for human consumption,
- documentation mechanism for programmatic (e.g. IDE) consumption,
- high-level process and functional description with concern for stateful resources,
- using http- and xml-based technologies, supporting for authentication and signing.

This document has adopted a notation for xml-rpc methods and types thus providing an elementary answer to the need of a human-documentation of components as web-services. The data-types used there are the same as SOAP *RPC-style* data-types so that a migration to this standard in its primitve form will not be an issue.

For notifications of state-updates, which is frequent in tracing user activity, ActiveMath has set up an event framework intended to propagate events through the application's components, offering subscription mechanisms and propagation at the local, remote, and client levels.

3.3 Component Integration

We shall distinguish three level of components:

The primary components are the pieces without which nothing can run in LeActiveMath. They will typically give birth to the rest of the components, or provide the basic services for the rest of the application.

Primary components are expected to be of use to most other components of Le ActiveMath. They are documented in their Java, xml-rpc, and event APIs.

Tightly integrated components include most of the components: they are living together with the core system, typically have no life without it, and use many of the services. A distribution of Le ActiveMath will include them.

Tightly integrated components should be documented in what functions they offer to the other components and which other components they use. Depending on the technical implementation and the expected other components using it, they will include a Java API, possibly an xml-rpc API, and events they generate or listen to.

Moreover, tightly integrated components may integrate into ActiveMath's web-based user interface. They will need to provide a Controller and an amount of views. For more information, refer to chapter 6 and especially section 6.4 for implementation details.

Loosely integrated components are components that have a life and configuration of their own and typically only *collaborate* with Le ActiveMath. They are integrated using little of the Le ActiveMath services and their integration is a coordination work, assortated with, possibly, a translation work.

Loosely integrated components should be best documented in the *processes* they intend to be part of in the collaboration in a similar fashion as the web-service choregraphy or BPEL standards.[1]

3.4 Security and Privacy

Le ActiveMath is a learning environment and, as such, needs to store and process information regarding the learner's competencies and past actions within the server. In itself, this is a privacy issue which requires, in principle, agreement of the learner (see [4]).

As a result, Le ActiveMath should prove itself secure against any unplanned access. Security mechanisms will be introduced to allow a learner's browser to be authenticated at each request. This authentication has to be propagated to other tools of Le ActiveMath.

Loosely integrated cooperating servers will have to trust each other. In a simplified world, this can be done based on IP-addresses.

Access to the web-services offered by Le ActiveMath are another possible vulnerability. Measures should be taken so that only the components of Le ActiveMath are accessed. Again, in a simplified world, one can trust the set of tightly integrated components, and IP-based restrictions are maybe enough.

Such restrictions, however, break the usage of web-services by client components (such as the assembling tool or the concept-mapping tool). For them, a relay mechanism will be written, which will allow clients with a witness of the authentication to access the services but will be able

[1] BPEL: Business Process Expression Language, see `http://www.oasis-open.org/committees/wsbpel/`.

to do so only being able to access the user-allowed information (including only the learner-model of this learner, and the content allowed to this learner).

Enforcing such a limitation may appear an exaggerated measure, it is, however, needed since the components on the client computer, once downloaded, are not anymore under control of the server deployment and may be affected by virusses or simple manual hacks.[2] Conversely, developing ad-hoc secured bridge accesses (such as dedicated servlets) for the elementary services needed by the client components would account to a fully separate development branch and would prevent any component re-usability.

[2]in a sense, spam is exactly using this form of abuse: it uses the SMTP servers *as if being anyone*.

Part I

Primary Components

This part describes the essential building blocks that form the basis of the ActiveMath system:

- ActiveMath Core
- ActiveMath Event Framework
- ActiveMath Front End
- ActiveMath User Management

4 ActiveMath Core

Responsible: Stefan Winterstein

4.1 General Architecture

4.1.1 MVC (Model-View-Controller) for Web Applications

One of the most famous patterns for layering user-interface software is the model-view-controller (MVC) architecture made popular by the Smalltalk community in the late 1970s [12].

Web-based Java user-interface architectures usually adopt the MVC approach and are divided into three separate sections: a browser-based user interface (the view), a set of Servlet-based controllers, and a number of services (APIs) wrapped around the application logic and model.

Web-based application logic is commonly triggered by an HTTP request performed using a URL coming from a user's browser, which invokes the controller class responsible for that URL with the parameters supplied. The controller implements the high-level application logic and, by invoking the APIs of the application, orchestrates the application services to access the desired information on the application's data model, and possibly perform modifications on it. The result is made of data objects, witnesses of the model, which are passed on from the controller to the view layer for display.

Details on how MVC is used in ActiveMath, what frameworks are used and how they are used for integration of Le ActiveMath components are described in section 6.

4.2 Component Management

ActiveMath being a servlet-based web-server, has its basic lifecycle entirely managed by the servlet-container ([5]). This means that, in some deployment-schemes, ActiveMath will be woken up and shutdown at the discretion of the servlet container.

At startup time, ActiveMath servlets are first created (by the servlet-container) which triggers the request for the ActiveMath services. The latter are generally available objects which are looked up by an interface name. Components can be created by servlet controllers or by services or by other components.

Destruction of the components happen at destruction of their services or servlets.

Configuration of the servlet container is done through its own mechanisms and the standard servlet web-application packaging mechanisms. These provide the ports listened to and, mostly, the association of the web-paths to the given classes.

Configuration of the components is done through a hierarchy of name-value pairs allowing components created by another to only have access to a part of the configuration. Access to the components' current configuration as well as provision of globally available services offer developers the possibility to write components which need little to no code about configuration management and thus allows most diverse deployment strategies.

In the ActiveMath Java Virtual Machines, a class wishing to be configured needs to have a constructor accepting a single parameter, a Subconfiguration object from where it will pull the needed configuration-information and services. Usage of this class has allowed the development of several components which could be used both in a heavy-use server-oriented setting and in a tiny-download applet.

ActiveMath configuration is done using properties files (see Javadoc of Properties class. This choice has several drawbacks but allows an server administrator to override a factory setting. This facility is fundamental when setting-up such a complex server. A notion of child properties is provided by the "." notation: `mbase.url` is, thus, a child of the subtree `mbase`. Using properties, one can also create lists by the usage of a *dumb child identifier*: for example the `xsl.imports.latex.*` properties produce, together, a list of XSLT files which need to be imported. The name standing after `latex` is irrelevant.

In order, ActiveMath reads the file `Manager.properties` (inside the code), the file `conf/ActiveMath.properties`, followed by `conf/ActiveMath-individual.properties`, the content-descriptors are then read (the files `conf/ContentDescr_*.properties`, finally, system-properties are evaluated. The last in this chain gives the final value which will be made accessible to the components as configuration.

System administrators thus wishing to adapt the system may do so adding property lines in `conf/ActiveMath-individual.properties`. They will read what to insert there by browsing and copying from `conf/ActiveMath.properties` which contains many explanatory comments.

4.3 Event Definitions

- `ApplicationStartup (Attr: none) [Tags: Application]`
  ActiveMath has been started.

- `ApplicationShutdown (Attr: none) [Tags: Application]`
  ActiveMath is being shut down.

5 ActiveMath Event Framework

Responsible for this component: Stefan Winterstein.

Components often want to be notified when events of interest occur in other services or components. Since the set of components interested in such messages is often unknown in advance or will change over time, a mechanism for registering interest and for propagatin events is needed.

Events (also called asynchronous messages) are a proven mechanism for a powerful and flexible, yet rather loose integration of components.

LeActiveMath provides events based on ActiveMath's lightweight event publication framework.

5.1 Event Basics

In the common terminology, a component can choose to *publish* events, making it an *event source*. Another component can *subscribe* to the events published by an event source, making them a *listener* (or event *consumer*), which will then receive *event messages* from the event source (sometimes also called *notifications*).

In contrast to a full-fledged messaging model, events are not sent from a specific sender to a specific recipient, but rather remain anonymous: when creating and publishing an event, the component is usually not aware who is listening to the events (only the module managing the subscriptions is). Also, the listener typically does not care which component or module created the event, it only knows where to subscribe to the events it is interested in.

The delivery model of events is typically limited to an asynchronous *push*, meaning that events are delivered from the source to the listener without waiting for the listener to react to the notification or return a result for it.

5.2 Event Objects

The minimum data carried by an ActiveMath event is:

- a **type**, indicating *what* happened (e.g. "user X has logged in", "new content available"). Depending on the type, the event may carry additional data for further describing the event.

  Event types must be unique in the same "event space", i.e. for all events published to an event publisher.

- a **timestamp**, indicating *when* it happened: Events happen at a single point in time (and space). The timestamp indicates the wall-clock time of the event source when the event took place. It is important to note that timestamps must be interpreted relative to the VM of the event source and are not to be used for a global ordering of events from other VMs without careful consideration.

- a **source**, indicating *where* happened. This is typically the name of the producer, i.e. the class name of the component that produced the event (e.g. `org.activemath.webapp.dict.DictionaryController`).

5.3 Event Tags

It is often desirable to group event types across several dimensions, e.g. for filtering ("I want all event concerning a user", "Give me all events related to the Dictionary"). However, events don't

fit well into a rigid type hierarchy: Is "user logged in" an application event? A user event? An interaction event? What's needed is a flexible, non-hierarchical type framework that allows for a mixin of characteristics such as "is associated to a user".

This is implemented by event tags. Tags are labels for event types. Each event type is associated to zero ore more tags. This relation is static, but allows for changes in the future.

Tag definitions can be nested, that is, tags can inherit from another. Multiple inheritance is explicitly allowed.

Tags can add attributes to an event. For example, the "user" event tag adds the attribute `userId` to any event tagged as such. Tags defined without attributes (such as "application" event tag) just serve as marker tags.

Figure 3 gives an overview of the event tags currently defined by the global Event Manager.

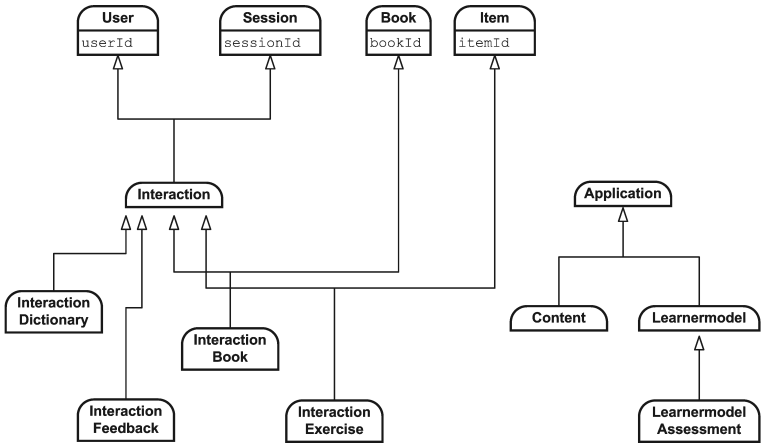

Figure 2: Event tag structure

User Event Tag Indicates that the event is tied to a specific user. This adds the following attribute:

- `userId` - id of the user that caused this event. Might be empty if user is anonymous (not logged in).

Session Event Tag Indicates that the event is tied to a specific web session. This adds the following attribute:

- `session` - id of the (web) session where this event occurred (a user can be logged in in serveral sessions).

Item Event Tag Indicates that the event is tied to a specific item. This adds the following attribute:

- `itemId` - id of the item. May be a content item or a domain-knowledge topic.

Book Event Tag Indicates that the event is tied to a specific book.

This adds the following attributes:

- `bookId` - id of the book.

Grouping Event Tag Indicates that the event is tied to a specific grouping.

This adds the following attributes:

- `groupingId` - id of the grouping.

Interaction Event Tag Indicates that the event is associated to a user interaction with the system.

Application Event Tag Indicates that the event is associated to the application (e.g. caused by the application, not directly by a user action).

5.4 Local Event Implementation

ActiveMath's event framework API is found in package `org.activemath.events`. See `http://www.activemath.org/~ilo/redirs/events.html` for a detailled and up-to-date documentation.

5.4.1 Event Types, Tags and Filters

Global ActiveMath event types are found in package `org.activemath.events.types`. All events inherit from `ActivemathEvent`. Event tags are associated by the use of interfaces.

Event tags are found in package `org.activemath.events.types`. They are implemented as interfaces. The `EventTag` class enumerates all tags and allows programmatic access to the tag classes. It also contains a method to match an event to a tag.

Event filtering is done through classed `EventFilter` and `EventFilterList` in package `org.activemath.events.impl`. An `EventFilter` can either match an event type or an event tag. An `EventFilterList` has support to add a filter based on type or tag.

5.4.2 Interfaces

Listener Interface (`ActivemathEventListener`)

- `onActivemathEvent(ActivemathEvent)` $\longrightarrow$void
  Handle an Activemath event.

Subscription Interface (`ActivemathEventSubscriptionService`)

- `subscribe(ActivemathEventListener)` $\longrightarrow$void
  Subscribes a listener to all events. Replaces a previous subscription of the same listener.

- `subscribe(ActivemathEventListener listener, EventFilterList include, EventFilterList exclude)` $\longrightarrow$void

  Subscribes a listener to selected events. Only events matching the include filter list and not matching the exclude filter list are delivered. Replaces a previous subscription of the same listener.

- `unsubscribe(ActivemathEventListener)` ⟶ void

  Unsubscribe a listener from his event subscription.

Publisher Interface (`ActivemathEventPublisher`)

- `publishEvent(ActivemathEvent)` ⟶ void

  Notify all listeners registered with this publisher of an event.

5.4.3 The Event Manager

The central event manager for ActiveMath is implemented by the class `org.activemath.events.impl.EventManager`. It offers a subscription and a publisher interface.

The Event Manager acts a central place for events of global interest. In case a component deals with events that are only of local interest (such as the learner model), they should implement their own, local event publisher.

5.4.4 Java Example

Here is an example for a local Java event listener:

```
class ExampleListener
    implements ActivemathEventListener
{
    ExampleListener()
    {
        // register myself for interaction events
        EventFilterList include = new EventFilterList();
        include.add(EventTag.INTERACTION);

        // ...but not for event type "UrlRequested"
        EventFilterList exclude = new EventFilterList();
        exclude.add("UrlRequested");

        EventManager.getInstance().subscribe(this, include, exclude);
    }

    public void onActivemathEvent(ActivemathEvent event)
    {
        if (event instanceof UserLoggedInEvent) {
            ...
        }
        ...
    }
}
```

5.5 Remote Events

It is foreseen that remote eventing will be needed in the future, for both server-to-server and rich-client-to-server communication. A special case is event exchange with the browser client. Here, we envision a solution based on asynchronous client-server using the browser's `XMLHttpRequest` object and an event encoding in Javascript.

5.5.1 XML-RPC API

ActiveMath will provide a central place to subscribe to events from its JVM. From here, remote components can recieve events of all event publishers inside ActiveMath's JVM.

5.5.2 Event Subscription

An EventPublisher web-service could to the following methods:

- `Subscribe(eventListenerUrl,referenceId,includeFilter,excludeFilter)` ⟶

  Subscribe to an event source by providing the callback URL of the event listener.

  The `referenceId` is sent with each notification message, it allows the management of further propagation. The event sink interface is described below.

  `includeFilter`, `excludeFilter` is used to specify the types of events the sink wants to include ore exclude in the subscription (eg. only events of specific types or tags). Details TBD, analogous to the Java API.

- `Unsubscribe(eventListenerUrl,referenceId)` ⟶

  Unsubscribe from an event source. The `eventListenerUrl` address and `referenceId` must match the one provided with the `Subscribe` call.

5.5.3 Remote Event Listener

A remote event listener has a single method:

- `onEvent(referenceId,remoteEventDetails)` ⟶

  Get notified of an event. The `referenceId` allows for identification of the publisher (where it was used for the Subscribe call). The event is represented in XML, see below.

5.5.4 Remote Event

`remoteEventDetails`
*

For simplicity we assume each events that is candidate to be exchanged will be encoded with the XML-RPC primitive and composite types. See 24.1.

5.5.5 Linearized Event Representation

When events are represented linearized in a file, an XML representation similar to the one below can be used:

```
<ActivemathEvent type="PagePresented" ts="1106913433214"
                 source="org.activemath.webapp.controller.ViewBook">
    <User id="Eva"/>
    <Session id="EAB428520EFF6570171766BC2A23513D"/>
    <Book id="analysisIndividuellComplete"/>
    <PagePresented page="1"/>
</ActivemathEvent>
```

Details TBD.

5.5.6 Example

The code using the remote event API from ActiveMath will look something like this:

```
  URL myServer = "http://amath.activemath.org:8888/EventManager";
  URL mySink = "http://localhost:7777/EventSink";
  String myRefId = "userEventSubscription";

  XmlRpcServer SubscribeRemoteEvents()
  {
    // obtain a reference to the server
    XmlRpcServer server = new XmlRpcServer(myServer);

    // Define event filter
    RemoteEventFilter includeEvents = new RemoteEventFilter();
    includeEvents.addTag("user"); // subscribe to user related events only

    // Subscription
    server.Subscribe(mySink, refId, includeEvents, null);
    return server
  }

  /* Notification of remote events.
   * The incoming XMLRPC call to "mySink"
   * must be linked to this method.
   */
  public void onEvent(String referenceId, RemoteEvent event)
  {
    if (referenceId.equals(myRefId)) {
      handleUserEvent(event);
    }
    ...
  }

  XmlRpcServer UnsubscribeRemoteEvents(server)
  {
    server.Unsubscribe(mySink, refId);
  }
```

6 ActiveMath Front End

Responsible for this component: Stefan Winterstein.

6.1 Description

As described in section 4.1.1, a Model-View-Controller (MVC) architecture serves as the basis for our web application. It separates application business-logic (*controller*) from application data (*model*) and the presentation of that data (*view layer*). It is well established for Java web applications, where servlets act as controllers, Java data objects form the model, and views are usually implemented by a dedicated template language.

Figure 3 gives an overview of ActiveMath's MVC architecture.

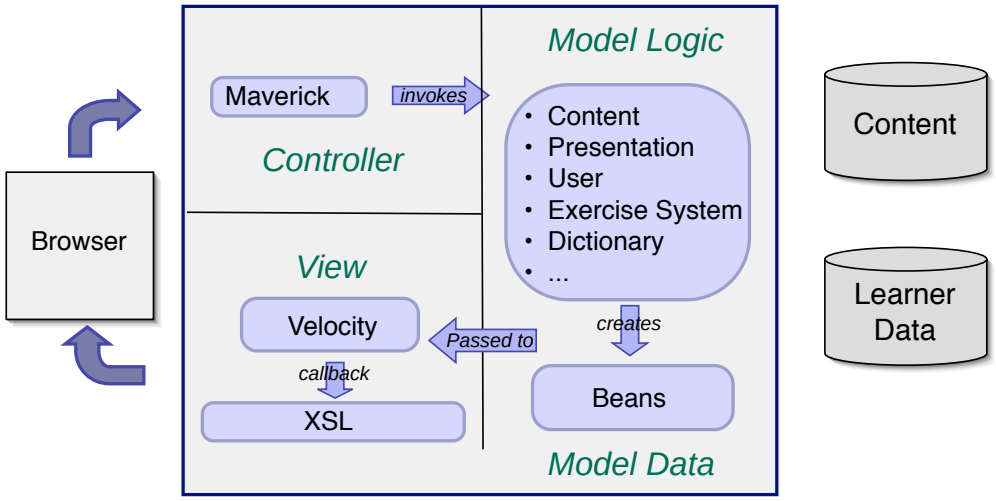

Figure 3: Overview of ActiveMath's MVC architecture

6.1.1 Web Frameworks in ActiveMath

The web application part of ActiveMath is based on two frameworks: Maverick and Velocity. Maverick (`http://mav.sourceforge.net/`) is a minimalist MVC framework for web publishing using Java and J2EE, focusing solely on MVC logic. It provides a wiring between URLs, Java controller classes and view templates.

Velocity (`http://jakarta.apache.org/velocity/`) is a high-performance Java-based template engine, which provides a clean way to implement the view-layer and incorporate dynamic content in text based templates such as HTML pages. It provides a well focused template language with a powerful nested variable substitution and some basic control logic.

6.2 Controllers

Controllers in ActiveMath are provided by the Maverick framework. They are based on standard Java servlets, but are instantiated by Maverick for each HTTP request. The framework automatically handles creation of the right controller class and passing the URL parameters to it.

All ActiveMath controllers must inherit from the common base class `org.activemath.webapp.base.ControllerBase`, which extends a Maverick controller. `ControllerBase` provides convenient methods for request and error handling, access to user and session information, the model and other vital functionality that simplify life for implementors of ActiveMath controllers.

Control is passed to controllers via the `action()` method. It doesn't take any parameters; instead they are passed bean-style by calling the appropriate setter methods, which is done automatically by Maverick.

The common task of a controller is to invoke the components of the application in a way that implements the desired functionality. It has to transform the incoming URL parameters into method parameters suitable for the components APIs, and handles the basic session management (like associating a request with a user). The results coming back from the model are wrapped into data objects suitable to display in the view layer.

These data-objects are "put into the model" under specific names that the view knows about and can be accessed there. They are inserted as "beans" (see next section). Controllers and its view(s) must agree on the model objects and their names.

In general, a controller does not implement much application functionality by itself, but is constrained to be a thin layer between the Web and the application.[3] Therefore, a controller can be viewed as a web-based (URL-)API of ActiveMath and shields the application from its Web interface.

6.3 The View Layer

The basic task of the view layer is to visualize the data objects ("model") coming from the controllers by providing a view on them. For our web application, the view is typically an HTML presentation, although ActiveMath is also capable of views in other formats, such as xHtML+MathML or SVG.

ActiveMath's views are implemented using the Velocity template language and can be recognized by the filename suffix `.vm`. Velocity implements a simple control language (such as `if/else`, `foreach`, `include`), macros, and a powerful way for nested variable references and substitution, which is used to create a view dynamically.

Data is provided to the view layer in the form of Java `beans`. The term "beans" in Java was originally used for the Java component model, but in this context stands for passive data objects that have publically accessible getter and setter methods.

Here's is an illustrating snippet in HTML, which greets the user by its name and lists some books that can be accessed:

```
<html><head><body>

    <h1>Hello, $user.Name!</h1>

    #phrase("your-books"):<ul>

    #foreach ($book in $books)
      <li> $book.Title
    #end

</ul></body>
```

[3] Although in reality, some controllers currently implement quite a bit of application functionality, e.g. the `DictionaryController`.

Note that

- `user` and `books` are beans that have been provided under these names by the controller
- a property of a bean can be accessed such as `$user.Name`. This will call the method `get` method of the model object (e.g. `User.getName()`).
- `#phrase()` is a special macro and will insert a suitable text in the user's language (see below)
- Velocity can loop over a collection object (`books`)

For further details, refer to the Velocity documentation (`http://jakarta.apache.org/velocity/`).

6.3.1 Internationalisation

ActiveMath provides several mechnisms for internationalisation of the Web front end that make it easy even for non-technical people to adapt the interface to another language.

Phrases Phrases are text snippets that do not appear directly in the view (e.g.. the HTML file), but are replaced by a call the the Velocity macro `#phrase(<key>)`. All phrases for a language are collected in a single file (`i18n/phrases_*.properties`) and given a unique key. Velocity then takes care of fetching the corresponding text from the phrase file of the desired language.

Phrases also provide a mechanism to specify parameters that are integrated in a phrase value, such as the user's name that must appear inside the greeting phrase. In this case, the phrase can contain a placeholder for one or more parameters (e.g. `"Hello, 0!"`, where the placeholder stand's for the user's name).

Language-specific filenames ActiveMath provides a transparent mechanism for all web-servable files: Filenames with a special language extension are given precedence over the original filenames. For example, when the original request is for `menu.vm` and the desired language is German (`"de"`), then the file `menu_de.vm` is served if it exists. If not, the original filename is used, of course. This mechanism applies not only to Velocity files, but also to most other files served by ActiveMath, like images, Javascript etc.

Language specific filenames are an alternative ways to translate a file. They tend to be harder to maintain since, a change in the (little) velocity-code they contain must be done in all files. However, these filenames are much easier to translate and allow a translator to see much better the context of the translation. For example, the same English sentence "Open Files" can be interpreted as an action or as a list...

Localized Beans For the ActiveMath front end, content is represented in the form of POJOs (plain old Java objects) holding the metadata (title, type, id etc). These objects are managed by the Content Manager (see section 8) and represent for example a single content item, a book or a TOC entry. The titles for these elements can be available in several languages. In order to customize these beans to the language of the current request (and to the current user), these data beans are passed to the `UserBeanFactory`, which takes care of converting them into beans suitable to be passed to the view layer. An item object, for example, is converted into a `UserItem` object whose title is in the language of the user's request.

Using the mechanisms described above, translation of the front end into a new language then basically consists of

- translating an existing phrase file into the new language
- translating language-specific Velocity files into the new language.

For more details, see the webapp file `i18n/I18n-Readme.txt`. Note that, currently, the xslt system used for display of the mathematical content (`OMDoc`) uses another mechanism for internationalisation that is beyond the scope of this section.

ActiveMath determines the language of a request from several sources, which are taken into account in this order:

1. URL parameter `lang`
2. User's preferred language (as set in the user's preferences)
3. Browser's preferred languages
4. ActiveMath's default language

6.3.2 Other features

Output formats The view layer of ActiveMath is further enhanced by the content transformation system based on xslt, which enables rendering `OMDoc` content for different output formats. This is done by the Presentation System, described in section 9.1.

Tool Beans The power of Velocity files is further enhanced by tool beans, which are Java objects that not only carry data (like usual model objects) but also provide additional functionality that can be called from the view layer. This is used for example by the Presentation System: the view has access to a `PresentationTool` bean which can be called from inside a Velocity file to do content transformation *while rendering the view*, for example, to generate the link to a dictionary view of an item. This allows for incremental rendering of the content and increases the percieved performance of ActiveMath considerably.

6.4 Integrating a Component GUI

Tightly integrated components that also need to integrate in ActiveMath's web GUI will need to provide their own controller. This sections sketches out the basic implementation steps; details are to be discussed with the ActiveMath development team.

The documentation of Maverick (`http://mav.sourceforge.net/`) and Velocity (`http://jakarta.apache.org/velocity/`) are required reading for this undertaking.

1. Settle on a short name for your component or functionality module. From here, decide on a package name where your classes will go, and on a directory for your Velocity templates. (These decisions need to be coordinated with the ActiveMath development team).

   For example, our module for user notes has `notes` as symbolic name, `org.activemath.webapp.notes` as package name, and `webapps/ActiveMath/notes` for the Velocity directory.

2. Put your component's classes into your package.

   For notes, these are the classes for the notes database, and the notes manager, which provides the notes API.

3. Write a controller class. It must inherit from `org.activemath.webapp.base.ControllerBase`. Action takes place in the `action()` method, which is the only one you need to write. `action()` will return the name of the view, say `"list"`. Your controller should produce some bean objects and put them into the model.

   Have a look at `NotesController` and other controllers to see what they look like and how the methods of `ControllerBase` are used.

4. Define your URL API, i.e. the URL parameters you want to use for your controller. Typically, a controller is associated with only one command (e.g. `note.cmd`) and takes any number of parameters. If it has several tasks to fulfill, introduce a parameter called `action` (e.g. `list`, `edit`, `save` etc.).

   For each URL parameter you need, write a corresponding setter method (eg. if your parameter is called `id`, write a `public setId(String)` method). Maverick will call it automatically and set the correct values (integer, boolean or collection values are also possible).

   For example, URLs for the notes controller look like `/ActiveMath2/notes/note.cmd?action=list&id=xxx`: `ActiveMath2` is our current application path, `notes` is the name you chose for your module, `note` is your command name, and `.cmd` is appended by Maverick.

5. Write a basic `Velocity` template file that will display the data generated by your controller. Have a look at the webapp file `notes/list.vm` for an example.

6. The wiring between your URL, controller and its views is configured in the webapp file `WEB-INF/maverick.xml`. For the notes controller, it looks like this:

```
<command name="notes/note">           <-- URL will be "notes/note.cmd"
  <controller  class="org.activemath.webapp.notes.NotesController"/>
                                                  ^-- controller class

  <view name="list"     path="/notes/list.vm"/>           <-- view 1
  <view name="edit"     path="/notes/edit.vm"/>           <-- view 2
  ...
</command>
```

6.5 Event Definitions

The ActiveMath Front End sends the following events:

- `UrlRequested (Attr: url, referrer) [Tags: Application, User, Session]`
  An URL has been requested. This is tracked by all controllers.

- `PagePresented (Attr: pageId) [Tags: InteractionBook]`
  User requests a page in a book.

- `ItemPresented (Attr: none) [Tags: Interaction, Item]`
  A content item is delivered to the user.

- `ItemSeen (Attr: duration) [Tags: Interaction, Item]`
  User has seen a content item for a certain duration (seconds).

- `Happiness (Attr: value) [Tags: InteractionFeedback]`
  Current degree of happiness (0..100) as indicated by user.

- `UserBookCreated (Attr: title, grouping, goals, scenario) [Tags: InteractionBook]`
  User has planned a new book.

- `UserBookRenamed (Attr: oldTitle, newTitle) [Tags: InteractionBook]`
  User is renaming a user book.

- `UserBookDeleted (Attr: none) [Tags: InteractionBook]`
  User has deleted a user book.

7 ActiveMath User Manager

Responsible for this component: Stefan Winterstein.

7.1 Description

The User Manager is a part of the ActiveMath core system and is not to be confused with the Learner Model.

In ActiveMath, we make a distinction between "user" and "learner": the term *user* is used in a general sense and stands for any person using the application. On the other side, a *learner* is a user *acting as* a learner, i.e. a learner is seen as *role*. Other user roles might be "teacher", "content author" oder "administrator".

As such, the User Manager does only deal with users in an application-neutral sense. It handles login and logout, manages user objects and acts as an event publisher (see section 5) for user events.

7.2 Java API

The user manager is implemented by the singleton `org.activemath.webapp.user.UserManager`, which provides the following public methods:

- `static UserManager getInstance()`: get a reference to the User Manager.
- `User login(userId)`: Logs in a user, creates and returns the `User` object. (Note: Authentication is currently handled outside the User Manager.)
- `void logout(userId)`: Logs out a user.
- `User getUser(userId)`: Return the `User` object for a user id. User must be logged in.
- `void registerNewUser(User)`: create a new user.
- `boolean hasRole(User, UserRole)`: say if a user is in a certain role (such as learner, teacher, author, visitor, admin).

7.2.1 The `User` object

The `User` object is a facade object that gives access to most user related data, such as

- user properties (id, name, e-mail, ...) and preferences (such as the preferred language)
- user books that have been planned
- book states (storing persistant information of last page viewed, first/last access, progress information)

For every user that is logged in, a `User` object is maintained by the `UserManager`. Controllers of the ActiveMath front end layer can easily get the correct user object of a request (via `ControllerBase.getCurrentUser()`).

7.3 Web-service API

TBD if needed (Tutorial Dialog? Open-Learner-Model ?)

7.4 Event Definitions

Here is an overview of events defined by the User Manager.

- `UserCreated (Attr: userName) [Tags: Application, User]`
  A new user has been created.

- `UserDeleted (Attr: none) [Tags: Application, User]`
  A user has been deleted from the system.

- `UserLoggedIn (Attr: remoteAddr, userAgent) [Tags: Application, User]`
  User has logged in.

- `UserLoggedOut (Attr: none) [Tags: Application, User]`
  User has logged out.

- `UserPropertyChanged (Attr: property, oldValue, newValue) [Tags: Application, User]`
  A property of the user has changed (e.g.. name, language).

Part II

Tightly Coupled Components

(each can be GUI-integrated with a dedicated controller and view) (each can use local or remote APIs)

8 Content Manager

Responsible for this component: Stefan Winterstein.

8.1 Description

The Content Manager creates and maintains representations of content objects.

Content objects are loaded from the available content repositories. Currently, this is only MBase, but will be extended to other repositories for LeActiveMath. The Content Manager will give unified and transparent access to all repositories.

The most basic content object is the content *item*, such as a definition, an example, an exercises or a proof.

Another important content object is the notion of a *book*. A book contains a nested table of contents (*TOC*), which refers to *pages*. A page then is a list of content items. Books handled here are *pre-recorded-books*, that is, books whose table of contents has been *manually* created and has been stored in the content repository.

Since content objects are stored in remote repositories, the Content Manager maintains a cache for item objects. This cache is crucial for the overall performance of ActiveMath, since access to content items is ubiquitous in the application. (A second cache level storing items transformed for display is maintained by the Presentation System, described in section 9.1.)

8.2 Java API

8.2.1 `ContentManager`

The Content Manager is implemented by the singleton `org.activemath.content.ContentManager` (obtained using the static method `getInstance`), which provides the following public methods:

- `getContentItem()` ⟶ `Item`
  fetches a content item from a repository. This will use the item cache.

- `getRecBooks()` ⟶ List(Book)
  gets a list of available static books ("recbooks").

- `getRecBook(bookId)` ⟶ Book
  : returns an object representing pre-recorded book. (Note: In general, books should be obtained via `User.getBook()`, which can also handle user-created books.)

- `getGroupings()` ⟶ List
  gets a list of available groupings, which are the basis for planned books.

- `getGrouping(groupingId)` ⟶ Grouping
  gets an object representing a grouping.

8.2.2 `Item`

`Item`
A class to represent a content-item that can, possibly, be presented and at least has metadata.

The basic object representing a content item in one of the repositories is `org.activemath.content.items.Item`. It is implemented as a bean object, providing access to the item's metadata, such as

- title, type, copyright
- flavors, difficulty
- relations ("for", "depends-on") and containing theory

Additionally, it can return several JDOM (document object model) representations of the `OMDoc`.

8.2.3 Other content objects

Other basic content objects are `Book`, `Page`, `Grouping`, `Toc` and `TocEntry`. See their java-documentation for detailled descriptions.[4]

8.3 Web service interfaces

To be done if needed (Assembling tool?)

8.4 Content Events

Here is an overview of content events that will be available from the Content Manager. (Note: Data that is available to an event type is indicated here as parameter. This does not imply that a function is being called, but is merely used for notational compactness.)

`ItemAdded (Attr: none) [Tags: Content, Item]`
A new item has become available.

`ItemChanged (Attr: none) [Tags: Content, Item]`
The content of an existing item has changed.

`ItemRemoved (Attr: none) [Tags: Content, Item]`
An item is no longer available in the repositories.

`BookAdded (Attr: none) [Tags: Content, Book]`
A new pre-recorded book is available.

`BookChanged (Attr: none) [Tags: Content, Book]`
The contents of a pre-recorded-book have changed (because items have changed inside the book or because the table-of-contents has changed).

[4] The java class documentation of ActiveMath is published at `http://www.activemath.org/~ilo/java-doc/`. Note, however, that many class-names mentionned in this document are actually hyperlinks to the corresponding java-documentation.

`BookRemoved (Attr: none) [Tags: Content, Book]`
A pre-recorded-book is no longer available.

`GroupingAdded (Attr: none) [Tags: Content, Grouping]`
A new grouping is available.

`GroupingChanged (Attr: none) [Tags: Content, Grouping]`
The contents of a grouping have changed (because items have changed inside the grouping or because the table-of-contents has changed).

`GroupingRemoved (Attr: none) [Tags: Content, Grouping]`
A grouping is no longer available.

9 Presentation System

Responsible for this component: Stefan Winterstein.

9.1 Description

The ActiveMath presentation architecture transforms OMDoc snippets into an output format which the client can display. It is used wherever `OMDoc` content needs to be displayed to a user:

- When viewing an ActiveMath book, the individual content items of a page have to be transformed into the desired output format.
- The Dictionary needs the Presentation System to display the content items
- The Exercise System creates `OMDoc` as an output and relies on the Presentation System to render it to the client.

Currently supported output formats are:

- HTML
- MathML
- LaTeX
- PDF
- SVG

9.1.1 2-staged Presentation Pipeline

The Presentation System implements a 2-stage approach for presentation that uses XSLT transformations combined with a template engine (Velocity).

Basically, the presentation pipeline, as shown in Figure 4 can be divided into two stages.

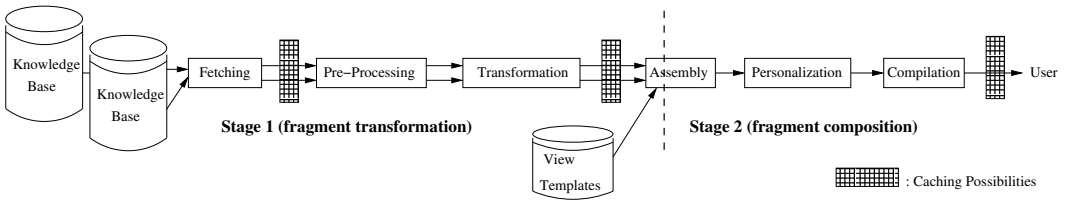

Figure 4: The presentation pipeline.

Stage 1: Transforming Content Items The first stage deals with individual content items, written in XML and stored in a knowledge base. At this stage, items do not depend on the user who is to view them. They hold unique IDs and can be handled independently. It is only in the second stage that items are combined to user-specific pages and enriched with dynamic data for this specific page request.

In Stage 1, the first part of the presentation pipeline is comprised of the steps *fetching, pre-processing*, and *transformation*. The *fetching* step collects requested content from the knowledge base. The output of this step are XML fragments. During *pre-processing* server-specific information

is inserted into the XML content. Then the *transformation* performs the conversion into the output format by applying an XSLT stylesheet to the document. The output of the last step are text-based content fragments, e.g. in HTML or LaTeX.

In order to improve performance, these fragments are stored in a cache, so that when the same page is requested a second time (or another page that contains the same content item) the expensive XSLT processing is not needed again.

Stage 2: Assembling a Personalized Page In Stage 2, these fragments are composed into a complete page and enriched with dynamic data for this specific page request.

Stage 2 performs the steps *assembly, personalization* and optionally *compilation*. The *assembly* joins the fragments together to form the requested pages. The fragments in the desired output format are integrated into a page template, which is fetched from an external source. During the *personalization*, request-dependent information is used to add personalized data to the document, such as user information or knowledge indicators. Another important personlisation is the translation of multi-language content into the requested language. If necessary, the *compilation* applies further processing to convert the generated textual content presentation into the final layout format. For example, PDF is generated by first assembling a LaTeXdocument, which is then converted into PDF using shell-based tools in the background.

9.2 Java API

9.2.1 Low-level API: Stage-1

The low-level API is accessible by creating `Formatter` objects with the appropriate parameters for output format and output language. It also takes care of invoking the XSLT transformations and caching the transformed items (called "fragments").

Among the methods:

- `formatItem(itemId, outStream)`: transforms a single content item to an output stream, using the output format, language and XSLT files configured for this Formatter. The transformation cache will be used for the fragments.
- `formatText(omdocString, outStream)`: same as `formatItem()`, but taking the OMDoc from a String, not from MBase.

9.2.2 View Layer API

Stage-2 of the presentation pipeline is integrated into the view layer of ActiveMath. It is accessed from within a Velocity file through the tool bean `TransformTool`. A `TransformTool` make use of a `Formatter` object to trigger Stage-1 of the pipeline.

Central methods are:

- `transformItem(itemId)`: This will render a content item to the stream of the associated `Formatter`. The rendering will pass all stages of the pipeline, including Velocity processing.
- `transformString(omdocString)`: Same as `transformItem()`, but taking the OMDoc from a String, not from MBase.

An even more comfortable way of using the power of the presentation system is the class `PresentationTool`. As the `TransformTool`, it is also a tool bean to be used from inside a Velocity template, but has the additional advantage of hiding the `Formatter` from the caller.

Also, with the `PresentationTool` it is possible to render the more complex output formats such as SVG and PDF, that would otherwise require special knowledge of the `Formatter`'s inner workings.[5]

9.2.3 Events

The presentation architecture listens to:

- `ItemChanged` which invalidates entries of fragment cache depending on the given item.

[5]For these formats, the output format used before the assembly stage is different from the final output format (e.g. PDF is produced by assembling the fragments in LaTeX).

10 Dictionary

Responsible for this component: Paul Libbrecht.

10.1 Description

The Le ActiveMath dictionary is a *knowledge browser* that presents OMDoc items extracted from an MBase one at a time and allows browsing along relations between them.

The Le ActiveMath dictionary is a component solely made for the learners hence offers no API for other components. It is mostly made a of a controller containing the whole logic to render the items' display and search-results.

10.2 Profile

10.2.1 Configuration and Lifecycle

It is woken up at first request. No configuration currently.

10.2.2 Java API

None.

10.2.3 Web-service API

None.

10.2.4 Event Definitions

Here is an overview of user events defined by the Dictonary.

`DictSearch (Attr: query,numResults) [Tags: InteractionDictionary]`
The learner has invoked the indicated query in the dictionary and the results were presented.

11 MBase

Responsible for this component: Paul Libbrecht

11.1 Description

The MBase is a storage for `OMDoc` documents. It is conceived as a generic storage for XML with limited query mechanisms (as opposed to XML-databases) but with good perfomances.[6] Query mechanisms are limited to:

- simple ID-based XML-content extraction using an ID-scheme customized to `OMDoc` with one granularity level (called an *item*)
- relations between items (which are declared only in one end)

11.2 Profile

11.2.1 Configuration and Lifecycle

MBase is offered to the Java virtual machine components as a service which is woken up at startup. Typically, however, this service may turn out to be a mere XML-RPC client of another server.

11.2.2 Java API

See `http://www.activemath.org/~ilo/redirs/MBaseRef.html`.

11.2.3 Web-service API

Strongly inspired by the Java API but adapted for XML-RPC. This communication channel makes no special effort to support large data-chunks (over 1Mb). A size which content items or relations are not expected to outpass.

Here are some of the methods offered by the XML-RPC interfaces of MBase:

- `getMetadata(itemId)` ⟶ `metadata-as-omdoc-content`

- `getTextualContent(itemId)` ⟶ `CMPs (a list of OMDoc content elements))`
  Returns a list of the CMP children of the item of given ID. CMPs are elements that contain the textual content in OMDoc.

- `getChildren(itemId xPathFilter)` ⟶ `children as OMDoc content`

  Returns the content of the element at itemId in the document database filtered by the xPathFilter expression. The latter is inspired by XPath but is smaller, it only allows either `.`, which means take all children, or <`elementName`> which means take all children with the given name (and drop the other elements).
  An example of such would be to request all children of all `code` children of a `omtext` element at MBase-ID "id". This can be done using `getChildren("id","code")`.

[6] Experiments show that about 300 queries per seconds are needed to serve a classroom. The current implementation satisfies more than this on a contemporary server.

- `getDependencies(itemId)` ⟶ `relations (as list))`
  Returns a list of relation objects, containing all relations originating from this item. See the description of the `itemRelation` type.
  Inverse of `getRelated`.

- `getRelated(itemIdlist of collections: strings)` ⟶ `relations (as list))`
  Returns the list of all relations that this item is the target of.
  Inverse of `getDependencies`. The MBase implementation is expected to pre-compute these relations as this method is expected to be fast.

- `getCommonName(itemId)` ⟶ record associating language-codes (e.g. "en") to title strings.

- `getType(itemId)` ⟶ type-string
  Returns the element-name of the corresponding `OMDoc` element.

XML-RPC types:

- `itemId`

  Parameters and return-values of this type are XML-RPC Strings using the encoding `mbase://collection/theory/name` as documented in ID class javadoc.

- `omdocContent`

  An XML-RPC string containing the OMDoc snippet requested. This string is the result of XML-parsing and re-outputting *as snippet* which implies that no DTD or pre-defined namespaces are needed in order to parse this snippet. Such a snippet is expected to have only one root element and, as such, be a well-formed XML-document.

- `itemRelation`

  An XML-RPC record with keys `of`, the origin of the relation, `on`, the target of the relation, and `type`, the type of relation. To be helped to remember it, think about *dependency of item A on item B*. The `type` of a relation is, generally, the value of the type attribute of the relation element. It can also be one of `ILLUSTRATES, PROVES, DEFINES, EXERCISES` when it reflects the relation provided by the `OMDoc for` attribute, `ABSTRACTLY_DEPENDS_ON` if it reflects the `dependson` relation, or `xx_IN_yy` for any other occurrence.

11.2.4 Events Emitted

`MBaseItemAdded (Attr: itemId) [Tags: Content, Item]`

The item itemId has been added into the MBase.

`MBaseItemChanged (Attr: itemId) [Tags: Content, Item]`

The item itemId has been changed in the MBase.

`MBaseItemRemoved (Attr: itemId) [Tags: Content, Item]`

The item itemId has been removed from the MBase.

11.2.5 Events Listened To

None yet (could be listening to other databases to update its external-reference set).

12 Tutorial Component

Responsible for this component: Carsten Ullrich

Todo: Extend with Suggestion Mechanism

12.1 Description

The tutorial component manages tutorial requests. It consists of two subcomponent, the tutorial control and the tutorial planner. The tutorial control receives tutorial tasks, selects which one to process and sends selected task to the tutorial planner. The tutorial planner generates sequences of learning material to fulfill a tutorial task.

12.2 Profile of the tutorial planner

12.2.1 Configuration and Lifecycle

The tutorial planner receives a tutorial task, a user identifier, a book identifier and a pedagogical strategy, and generates a sequence of learning materials that resolves the task, according to the standard or a given strategy. Only the tutorial component calls the tutorial planner directly, all other components call it indirectly via the tutorial component. The tutorial planner uses information from the learner model and from content repositories.

Configuration is done through the tutorialComponent.* properties.

12.2.2 Java API and web-service interfaces

- `getCourse( userID bookID tutorialTask)` ⟶ `extendedBook`

  Returns a grouping that provides an answer to the tutorialTask with respect to the standard strategy.

- `getCourse( userID bookID tutorialTask tutorialStrategy )` ⟶ `extendedGrouping`

  Returns a grouping that provides an answer to the tutorialTask with respect to a given strategy.

It is still an open question whether information about how to linearize the content is necessary at this point.

XML-RPC types:

- `userID`

- `bookID`

  The bookID is a character-string providing a unique-identifier of a book. Books are managed (stored, created, ...) by the content manager, see Section 8.

- `tutorialTask`

  An XML-RPC Record representing the tutorial command that the tutorial planner should generate content for (key: `command`) and an array of its parameters (key: `parameters`). A list of possible tasks will be made available. Example: `teachConcepts(`c_1`,...,`c_n`)`.

- `extendedBook`

  An Extended Book is a standard book that can contain tutorialTasks and requests for narrative bridges.

- `tutorialStrategy`

  An XML-RPC String representing the tutorial strategy that the tutorial planner should use. A list of possible strategies will be made available.

12.3 Profile of the tutorial control

12.3.1 Configuration and Lifecycle

The tutorial control manages requests of presenting content. These request can come from the learner and Le ActiveMath components such as the suggestion mechanis and the Open Learner Model. It sends the selected request to the tutorial planner and presentes the generated content.

Configuration is done through the tutorialComponent.* properties.

12.3.2 Web-service API

- `getPreferredStrategy(userID bookID)`$\longrightarrow$`tutorialStrategy`
  This method returns the strategy used to build the book. It is used by the dialogue manager to tune the type of feedback that will be applied.

12.3.3 Events

- `tutorialRequest (Attr: userID,bookID,tutorialTask) [Tags: tutorialComp]`

  When the tutorial control receives this event, it processes a given tutorial task with respect to a standard tutorial strategy.

- `tutorialRequestWithStrategy (Attr: userID,bookID,tutorialTask,tutorialStrategy) [Tags: tutorialComp]`

  When the tutorial control receives this event, it processes a given tutorial with respect to a given tutorial strategy.

- `tutorialRequestPlanned (Attr: tutorialTask) [Tags: tutorialComp]`

  When the tutorial control sends this event, then a plan was successfully generated that fulfills the tutorial task.

- `tutorialRequestPresented (Attr: tutorialTask) [Tags: tutorialComp]`

  When the tutorial control sends this event, then the Tutorial Task has been presented to the user.

12.4 Requests of the tutorial component to other components

12.4.1 Metdata queries.

The tutorial planner will send requests to the metadata query engine. See Section 14.1 for a description of the requests.

12.4.2 Learner model.

The tutorial planner has to have access to all instructionally relevant information about the user which includes her current knowledge state, her preferences, her history, her learning/cognitve style, misconceptioms, static profile, traits (what exactly needs to be requested by the pedagogical experts) etc. Queries: What is the mastery of Anton of "average_slope"? What is the cogntive style of Anton? Does Anton know how to work with Omega?

12.4.3 NLG

An `extendedBook` returned by the tutorial component contains requests for narrative bridges (see the use case in Section 15.5).

12.5 Use Cases

12.5.1 Course Generation

Players

- Metadata query engine
- tutorial control and tutorial planner
- User model

Description A request for course generation is send to the tutorial control. The tutorial control call the tutorial planner, which starts the course generation. During planning, the tutorial planner send queries to the query manager about content metadata, e.g., "Is there an easy example for computer science students for the concept average_slope". Additionally, it queries the user model, e.g., "Has the user already seen the exercise with id `xxx`?". The result of the course generation is a table of content, possibly annotated with additional information such as time and navigation constraints.

Exceptions If no content is available that fulfills the request, an appropriate message has to be passed to the learner.

12.5.2 Request for Supplementary Content

Players

- Metadata query engine
- Tutorial component
- User model
- Requesting "component": learner, suggestion mechanism, ...

Description The learner works on a course and himself or another component (e.g., the suggestion mechanism, the open learner model, ...) sends a request for additional information to the tutorial control. In case the tutorial control receives several requests, it selects the one to process first. This request is send to the tutorial planner. The result (a sequence of learning material) is presented to the learner and on demand inserted in the original course. The tutorial control sends a message that the request was fulfilled.

Exceptions If the request cannot be answered, an appropriate message has to be passed to the requesting component.

13 Suggestion Component

Responsible for this component: Carsten Ullrich

13.1 Description

The suggestion mechanism is responsible for the generation and presentation of global suggestions and hints. It consists of three main subparts. The diagnostic subcomponent infers interesting aspects of the learner's interactions with the system. The generation of appropriate feedback is performed by the suggestion engine. The suggestion engine determines the tutorial actions (tasks) to be taken in response to the needs and calls the tutorial component to execute these tasks. The suggestions are rendered and presented by the rendering subcomponent.

13.2 Profile

The suggestion mechanism gets notified of the learner's interactions with the system and generates appropriate feedback.

13.2.1 Configuration and Lifecycle

The suggestion mechanism's lifecycle is closely related to a learner's session: the suggestion mechanism is started and attached to a new session when it begins, and it is destroyed when the session ends.

13.2.2 Java API and Web-service interfaces

- `generateSuggestionFor( userID )⟶Suggestion`

  This can be used by other components to have the suggestion mechanism try to generate a relevant suggestion for the given user.

- `onEvent( Event )⟶void`

  This can be used by other components to notify the suggestion engine of new events.

14 Metadata Query Engine

Responsible for this component: Carsten Ullrich

14.1 Description

The metadata query engine serves as an access gate for ActiveMath and non-ActiveMath components to one or several content databases. Clients of this engine are, e.g., the tutorial component, the new dictionary, possibly the learner model. The servers are MBase and the exercise repository.

The query manager offers two services. It provides a single interface to a large number of databases, and it translates among several knowledge representations.

A typical query to the metadata query manager inquiring about the existence of an element (existence queries) has the form "Is there an element with the following properties...". An answers consists of a list of identifiers (strings) of those elements that fulfill the query.

The following examples are queries from the tutorial component. They integrate information about

- the class of an element (e.g., exercise, definition);
- the properties of an element (e.g., type, difficulty, learning context, depends-on);

The queries do not include information about the user. This is a separate functionality which is to be offered by a different component.

Existence queries have the form `getItem` *`constraint`*, with *`constraint`* being one or several conditions about the class and properties. Value queries have the form `getValue id (property prop)`. The values for the classes and properties are taken from the ontology of instructional objects ([13]).

- Return a definition for the concept *xxx*:

  `(getItems (class definition) (property for` *`xxx`*`))`

- Return all easy exercises for *def1*:

  `(getItems (class exercise) (property for` *`def1`*`) (property difficulty low))`

- Return all easy computer science exercises for *def1*:

  `(getItems (class exercise) (property for` *`def1`*`) (property difficulty low) (property field computer-science))`

- Return the concepts that the concept *slope* is based on:

  `(getItems (class concept) (property IsBasedOn` *`slope`*`))`

- Return the concepts that depend on the concept *slope*:

  `(getItems (class concepts) (property IsBasisFor` *`slope`*`)`

- Queries involving multiple dependencies: Return a real-world problem that is for the concepts *slope, difference_quotient*, and *differential quotient*:

  `(getItems (class real-world-problem) (property for` *`slope`*`) (property for` *`difference_quotient`*`) (property for` *`differential_quotient`*`))`

A third kind of query involves linearization. While the list of identifiers returned by the method `getItems` are not ordered, the query `(getSortedItems` *`constraints`*`)` returns an ordered list of identfiers. This approach could be generalized by allowing a constraint `(order` *`property-name`*`)` that orders the list of identfiers according to the order of the property, if an order is defined (e.g., for difficulty).

Queries about values of attributes are handled by the content manager (Section 8).

14.2 Profile of the Metadata Query Engine

14.2.1 Configuration and Lifecycle

Configuration is done through the metadataQueryEngine.* properties.

14.2.2 Java API and Web-service interfaces

- `query(constraints)`$\longrightarrow$`itemId`
  Sends a query to the metadata query engine checking for the existence of elements that fulfill the conditions.

XML-RPC types:

- `constraints`

  Constraints is a list of strings having the form

15 Dialogue Manager

Responsible for this component: Claus Zinn

15.1 Description

The dialogue manager (DM) offers two forms of dialogue: "local tutorial dialogue (within an exercise)" (DoW: D5.2), and dialogue about the Open Learner Model (DoW: D5.3). The DM interacts with a natural language front-end to verbalise its (symbolically represented) feedback moves. The NLG component can also be accessed by other components of LeActivemath, in particular, by the TC for the generation of course overviews, introductions and narrative bridges. Fig. 5 depicts the architecture for LeAM that we have in mind (as a rough guide).

15.2 Profile

15.2.1 Configuration and Lifecycle

No configuration is anticipated. The DM will be invoked for two purposes: managing natural-language tutorial dialogue and managing dialogue about the OLM. After each dialogue, the DM terminates itself.

15.2.2 Java API

None. All communication between the DM and other LeActiveMath components is done via web-service requests.

15.2.3 Web-service interface

- `performExercise(student,exerciseId)` ⟶
  . Triggered by the LeActiveMath event framework. This will invoke the DM for performing natural-language tutorial dialogue within an interactive exercise. For the interaction between the ES and the DM, the following three messages will prove useful:
- `student_chat_input(+englishString)` passing the student's current turn input from the ES console to the DM.
- `student_workarea_input(+englishString)` passing the student's editing on the work area from the ES console to the DM.
- `student_math_input(+mathExpression)` passing the student's editing on the math expression part of the work area of the ES console (say, with the formula editor) to the DM.

- `performOLM(+student, +exerciseId)` ⟶.
  Triggered by LeActiveMath event framework. Will invoke the DM to perform a dialogue about the OLM.

- `verbalise(requester, textType,textRepr)` ⟶
  . Will invoke the NLG component of the DM to verbalise a symbolic representation `text` of some type `textType`. The exact representation `textRepr` has to be fleshed our in more detail; this can be a sequence of *dialogue moves*, together with *discourse relations* holding between such moves. The `textType` can be one of `motivation, introduction, bridging, summary` or `directive` (preliminary list).

15.2.4 Events

None. All communication between the DM and other LeActiveMath components is done via XML-RPC queries.

15.2.5 Architecture

We anticipate the following communication between the DM and other LeActiveMath components, see Fig. 5.

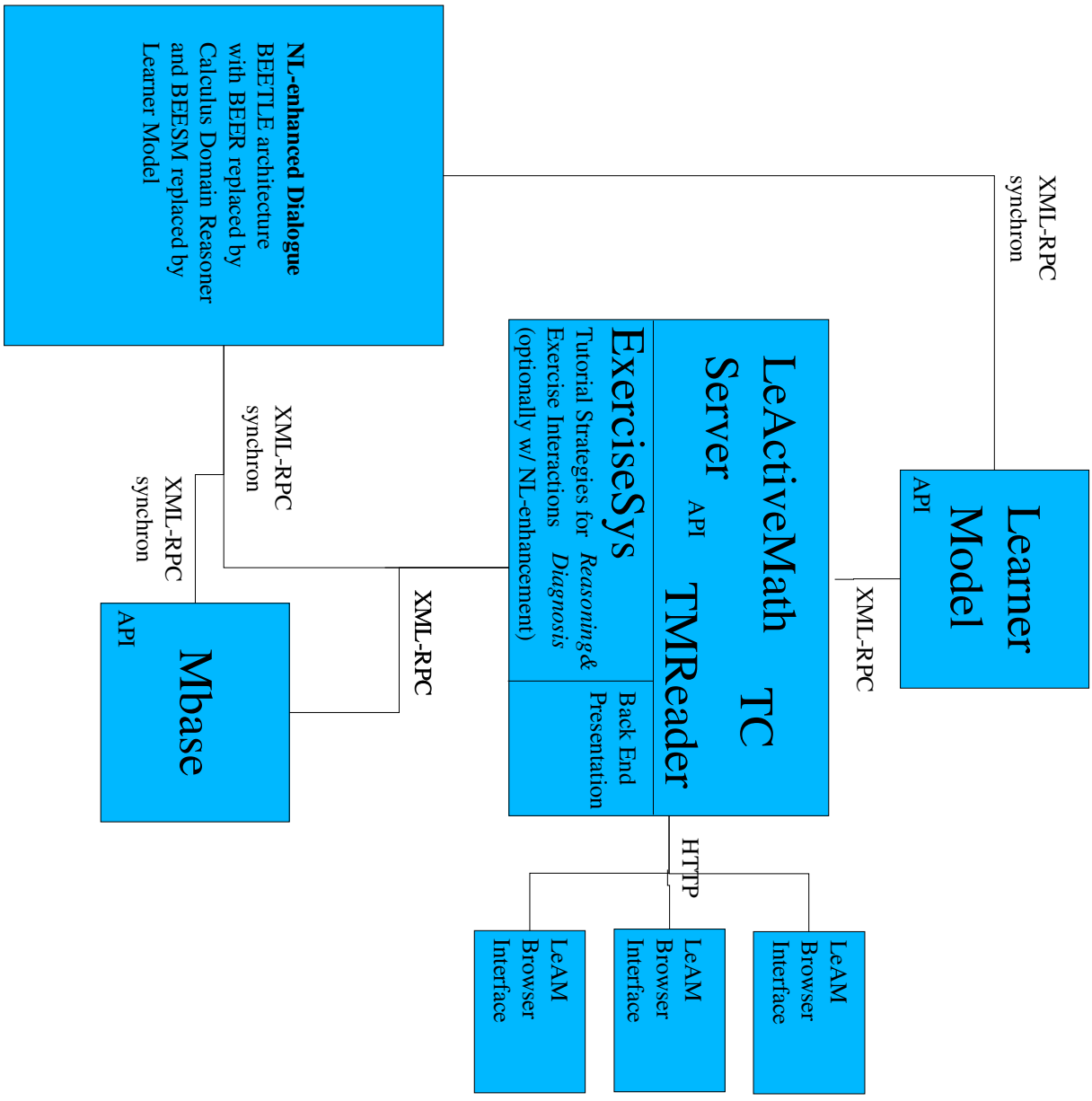

Figure 5: A *preliminary* view of the overall LeAM architecture.

15.3 Use Cases I: Local Tutorial Dialogue

15.3.1 Starting the DM — Initialisation

Because of a student clicking on a link for an interactive exercise, the LeActiveMath event framework "informs" the Exercise System (ES) to "perform" exercise `ex_id` with natural-language support. Then, the ES checks whether the appropriate knowledge sources (KS) are available to perform `ex_id` *without* NL-support. If the required KS are available, the ES sends a message to the DM to perform `ex_id`. Then, the DM checks whether the appropriate KS are available to perform `ex_id` *with* NL-support. Note that KSs may include available dialogue strategies given overall pedagogical settings, appropriate reasoning and diagnosis services, and linguistic resources. The DM check is successful if a dialogue initialisation phase succeeds, and an initial dialogue plan has been generated. For this initialisation phase, the following components (and others) will be queried by the DM:

TC: `getPreferredStrategy`(+student, +book, -strategy), where `strategy` *could* be `didactic`, `Socratic`, `PVE` *etc.*, and `book` is an identifier referring to some curriculum material (e.g. product-rule).

LM: `getOverallStudentPerformance`(+student, +topic, -performance), where the student's overall `performance` for some topic (or book) is `excellent`, `good`, `average`, `poor` *etc.* Note that this message is different from `getBeliefs`(+student, +domain, +topic, -belief), which returns the learner model's belief about the student's belief about a *topic* (or *concept*, or *skill*).

MBASE: `getMetadata`(+ex_id, -ex_description), where `ex_description` is the representation of the exercise (its metadata), say for the time being, restricted to the problem statement (but may contain some meta annotation like `difficulty`).

DR: `getSolutionGraph`(+exercise_description, -solutiongraph), where `solutiongraph` contains various ways (to be detailed) to solve a given exercise. *Note that this depends on the design and implementation of the domain reasoning and diagnosis component, which is not part of this document.*

There are calls to other components, but those components are part of the DM only; these internal modules are not accessible by other LeAM components (*e.g.*, linguistic resources, pool of strategies).

15.3.2 Performing the dialogue

If the initialisation phase has been successful, the DM needs to communicate via the ES to display tutor feedback and to get student input forwarded from the chat/interaction console. The DM will also share resources for reasoning and diagnosis (*i.e.* those resources that the ES uses to perform non-NL system-student interaction).

For the communication of tutor feedback to the student, the following services will be required:

NLG: `verbalise(+dialogue_move_seq, -englishString)` asks the natural language generation component (available for other modules than the DM as well) to verbalise a given symbolic representation of dialogue moves into English (Note that the NLG may want to query other components to inform its lexical/syntactical choice, which we omit discussing in this document. Also note that the resulting verbalisation may mix text with formulae (multi-modal feedback), and this may generate multiple/appropriate calls to the ES).

ES: `display(+englishString)` asks the ES to display a given tutor feedback in English on the console. We anticipate that the ES passes on this request to the presentation component of LeActiveMath.

ES: `display_math(+mathExpression)` asks the ES to display a given mathematical expression on the console.

ES: `highlight_math(mathExpression)` asks the ES to highlight a given mathematical expression on the console or workarea. Note that highlighting will be restricted to mathematical expressions in the current tutorial feedback turn, or to parts of the previous natural language dialogue.

We omit further discussing multi-modal feedback here, for the time being; much depends on the design of the student's work area, the math input editor *etc.* There will be a larger set of messages, where the DM tutor asks the ES to supply information to a graphical representation of the exercise's solution, to point to terms and subterms *etc.*

We may also require mixed expressions (i.e. combinations English strings and mathematical expressions) to be passed to the DM. (There is also question, of course, of what constitutes a turn.)

For the analysis of student input, the following services will be required:

DR: `diagnose_input(+studentInput, -diagnosis)`, where a semantic interpretation of the student's input is passed on to the diagnoser to check for correctness, misconceptions *etc.*

LM: `feedDiagnosis(+topic, +ex_id, +diagnosis)` passes the diagnosis on to the LM for the actualisation of the learner model. (Note, this means that the LM is continuously (and asynchronously) updated with student performance information while the student attempts an exercise).

The interaction between the ES and the domain reasoning and diagnosis engine has to be specified in more detail. For example, it might involve querying the LM about a given's student characteristics (say, the misconceptions that the system attributes to this student).

For the generation of tutorial feedback, the following services will be required:

LM: `getSummaryBeliefs`(+student, +domain +concept, -belief), where `belief` $\in [0,1]$ is the system's belief about the degree of knowledge of the student regarding a particular `concept` or topic (*e.g.* the product rule).

LM: `getSkill(+student, +domain +concept, -skill)`, where `skill` $\in [0,1]$ is the system's belief about the skill of the student regarding a particular procedural `concept` (*e.g.* differentiation of quotients).

LM: `getOverallStudentPerformance(+student, +topic, -global_performance)`. *Assuming that the LM is capable of performing such kind of global value judgement.*

LM: `getLocalExercisePerformance(+student, +topic, -local_performance)`

LM: `getValueSituationalFactor(+student, +sf, -value)` returns the value of situational factor `sf`.

LM: `getAutApp(+student, -aut, -app)` returns autonomy and approval for a given student.

MB: `getDefinition(+concept, -conceptDefinition)` asks MBASE to return the definition of a concept, potentially with meta-annotation. *This may use* `getTextualContent`*, if such actual content can be usefully employed.*

15.3.3 Terminating the dialogue

When the dialogue has come to an end (the exercise has been solved; the tutor or the student has chosen to abort), the control passes back from the DM & ES to the component that lead to their activation.

? `ExerciseFinished(+ex_id, +status)` informs the LeActiveMath component that activated the ES that the exercise `ex_id` has been terminated with status `status`, one of `success`, `student_abort`, `tutor_abort`. *A similar message/event may need to be sent to the LM and other interested parties.*

Also see `http://www.activemath.org/~ilo/javadoc/org/activemath/webapp/user/events/ExerciseFinishedEvent.html`.

Note that student performance throughout an exercise will be continuously monitored via the `update_lm(+ex_id, +diagnosis)` messages described above. Note also that for the generation of tutorial feedback, the LM will be continuously queried wrt. student performance and beliefs.

15.4 Use Cases II: OLM dialogue

OLM dialogue is "dialogue about the open student model to help LeActiveMath diagnose the student's state with respect to different facts and skills" (DoW, page 37). We will specify such dialogue when the specification of LM and the parts that will be accessible to the student (OLM) become much clearer. At the moment, it is unclear what the LM contains, and what contents should be made accesible with the OLM; so it is unclear what should be talked about in OLM dialogue.

15.5 Use Cases III: NLG

The NLG part of the DM will provide a NLG front-end for LeActiveMath's tutorial component to provide course overviews, introductions, and narrative bridges between learning objects. In a July meeting (CU, CZ), the following tasks have been discussed:

- Motivational text with learning goal

  "In this section, you will learn something about the differentiation operator"

- Introductory text:

  "First, we look at a concrete problem:..." or "Before giving the definition of X, have a look at the following problem:"

- Bridging text:

  "Now, that we are familiar with the problem, we discuss how it can be modelled using the concept of X"

- Summary text:

  "In this section, we have studied the main concepts and uses of the differential operator"

  (probably with mentioning METHOD, e.g., repetition, exercise, drill)

- Directives:

  "Please make yourself familiar with the following concepts: X, Y, Z. You should use the lexicon!" or "Look-up the necessary prerequisite knowledge in the dictionary"

This is highly preliminary, of course. NLG realisation strongly relies on an appropriate representation (curriculum, learning objectives, etc). We need to get familiar with such representations before being more specific.

16 Learner Model

Responsible for these components: the WP4 group: Paul Brna (UNN), Rafael Morales (UNN), Nicolas van Labeke (UNN), Éric Andrès (DFKI), Kaśka Porayska-Pomsta (Edin), Helen Pain (Edin).

The following components are described here: Learner Model (LM), Situation Model (SM), Open Learner Model (OLM) and Learner History (LH). They are grouped together under the name of *Extended Learner Model (XLM)*, in order to distinguish the group from the Learner Model component itself.

Responsible for this components: Eric Andrès Paul Brna, Rafael Morales, Kaśka Porayska-Pomsta and Nicolas van Labeke.

16.1 Description

As said above, the Extended Learner Model is composed of the following components:

Learner History (LH) This component manages the storage and retrieval of all historical information that is regarded as relevant for learner modelling.

Learner Model (LM) This is the component responsible for maintaining a faithful representation of the state and traits of every individual learner, which is used by Le ActiveMath to adapt learning experiences to each learner's needs.

Open Learner Model (OLM) This component provides learners with access to their learner models in Le ActiveMath, and includes facilities for learners to negotiate their models.

Situation Model (SM) This component is responsible for capturing aspects of the situational context which are relevant to tutors adapting their feedback to the individual cognitive and affective needs of the students.

The description of the interface of the Extended Learner Model is based on the proposal for a fragment of Le ActiveMath architecture shown in figure 6. Information flow depicted in grey is internal to XLM and hence it is not described in this document—see [7] for further information on the proposed architecture, and [6] for documentation of XLM internal information flow. In the figure, and in the rest of section 16, Domain Knowledge (DK) stands for the set of Le ActiveMath components in charge of describing the subject domain and teaching-learning material, such as the Content Manager (CM), MBase, Siette, the Exercise Repository (ER) and any Domain Reasoner (DR). XLM acts as client of DK, whilst it acts as server and client of the Dialogue Manager (DM), the Tutorial Component (TC) and the User Interface (UI).

16.2 Conformance

16.2.1 Satisfied requeriments

The interface to the extended learner model described in section 16.4 supports the use cases described in section 16.3 and meets the Le ActiveMath requirements [9] listed below.

Creating a learner model (section 16.4.2) Meets requirement 3.5.

Updating a learner model (section 16.4.2) Meets requirement 3.5.

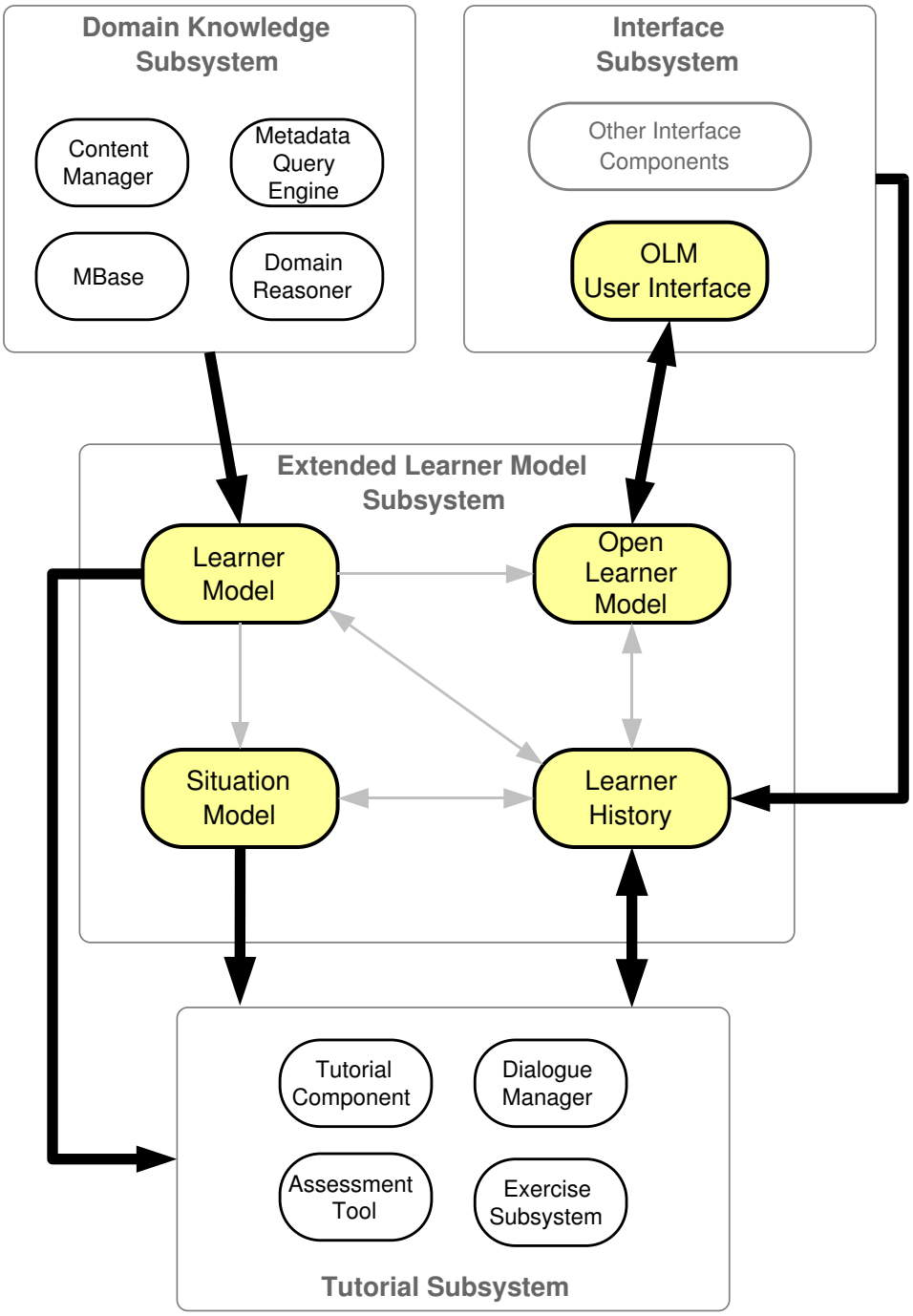

Figure 6: Fragment of Le ActiveMath architecture, in the surroundings of the Learner Model.

Exposing a learner model (section 16.4.2) Meets requirements 3.5, 4.18[7] and 6.2[8].

In addition, the interface to the extended learner model supports satisfaction of requirements 4.13, 4.17, 4.19, 5.9 and 7.3.

Note that, being this section devoted [up to now] to describe the interface to the extended learner model, requirements on its contents and internal information flow do not apply. This is the case, in particular, of requirements 5.1, 5.2 and 5.4.

16.2.2 Assumed requirements

The interface to the extended learner model described here assumes satisfaction—by other Le ActiveMath components—of the requirements listed below.

Creating a learner model (section 16.4.2) Assumes satisfaction of requirement 4.15[9].

Updating a learner model (section 16.4.2) Assumes satisfaction of requirements 2.3, 2.5, 3.6[10], 4.21, 4.22, 5.3, 6.5, 7.4, 7.5, 7.7 and 7.8.

16.3 Use Cases

16.3.1 Creating a learner model

Rationale The learner model of any learner registered in Le ActiveMath will be composed of a domain dependent section and a domain independent one. In the first section there will be information such as beliefs on the learner's knowledge of a topic or the learner's mastery of a competence. In the second section there will be information about the learner such as beliefs on learning style and self-confidence, which can be regarded as domain independent. It is better design, and increases the flexibility of LM, to keep the *definition of the structure of the domain* (DSD) in DK, where knowledge about the subject domain is kept, and to retrieve it (if necessary) when creating a new learner model.

Description

1. XLM gets a request from a component (to be specified, but probably Le ActiveMath Core).
   - Learner identifier.
   - Preferences and other traits.
   - Domain identifier.
2. XLM sends a request to DK.
   - Domain identifier.
3. XLM gets a response from DK.
   - Domain description (e.g. ontology, concept map).

   3.1. XLM initialises the model for the given learner.
4. XLM confirms model creation.

[7] XLM provides the needed access to instructionally relevant information about the learner.
[8] XLM provides the needed access to its components.
[9] Actually, it assumes the existence of an ontology of learning objects.
[10] XLM requires the reverse functionality; that is, that given a set of learning object identifiers, DK can the corresponding metadata set.

16.3.2 On-line updating of a learner model

Rationale Continuous updating of beliefs in a learner model feeds from, and it is triggered by, information of new events made available. However, it may be necessary to complement this information with further information about the learning objects a learner has been interacting with, which would need to be requested from DK.

Description

1. XLM gets an event related to a learner.
   - Learner identifier.
   - Description of event.
2. XLM sends request to DK.
   - Learning object identifiers.
3. XLM gets a response from DK.
   - Metadata.

   3.1. XLM updates its beliefs accordingly.

16.3.3 Off-line updating of a learner model

Rationale There are situations in which XLM may need to take the initiative to update a learner model. For example, XLM may need to be synchronized after some maintenance work; there may be some events to which XLM does not react immediately, but only after a critical amount of them have been accumulated, etc.

In this situations, XLM will initiate the process, which will continue as in the previous use case (section 16.3.2).

Description

1. XLM sends request to DK.
   - Learning object identifiers.
2. XLM gets a response from DK.
   - Metadata.

   2.1. XLM updates its beliefs accordingly.

16.3.4 Fully exposing a learner model

Rationale The full content of a learner model may be needed by other components of Le ActiveMath such as DM and TC. Other possible use is for exporting a learner model (e.g. to be used in another instance of Le ActiveMath).

From the same perspective as the use case of creating a learner model (section 16.3.1), it is assumed that XLM may store learner models for more than one domain, hence a domain identifier is considered besides a learner identifier.

Description

1. XLM gets initial request from DM or TC.
   - Learner identifier.
   - Domain identifier.
2. XLM answers.
   - Complete learner model (for that domain).
   - Calculated autonomy and approval values.

16.3.5 Partial exposition of a learner model

Rationale Depending on the belief updating algorithm employed by XLM, a single event may trigger changes to several beliefs in a learner model, many of them not relevant to next decision making by other components in Le ActiveMath. Hence, along a learning session and as a learner model evolves, access to specific beliefs may be more convenient than requesting complete copies of it.

Description

1. XLM gets a request from DM or TC.
   - Learner identifier.
   - Domain identifier.
   - Topic identifier(s) or request for autonomy and approval values.
2. XLM answers.
   - Belief(s) on topic(s) or calculated autonomy and approval values.

16.3.6 Partial exposition of the Learner History

Rationale It might well be the case the some components wish to access information regarding the learner's history. The Tutorial Component, for instance, will want to know what a learner already did with a given item before choosing an appropriate strategy to present it (see section 12.5.1). Similarly, the Dialogue Manager may also be interested in certain actions the learner performed. The components will issue a query to the Learner History to find out if a specific interaction has taken place.

Description

1. XLM gets a request from DM or TC.
   - Learner identifier.
   - Topic(s).
2. XLM answers.
   - Interaction(s) on topic(s).

16.3.7 Partial exposition of a known part of the Learner History

Rationale During negotiation of the Learner Model content, the Open Learner Model might need to justify or explain the current state of the beliefs. In order to do that, it has to access precise interactions that are known to have contributed to the current state of the belief. This way of accessing the Learner History might prove useful also for other components like the Dialogue Manager or the Tutorial Component.

Description

1. XLM gets a request from DM or TC.
   - Learner identifier.
   - Interaction identifier(s)
2. XLM answers.
   - Interaction(s).

16.3.8 Inferential diagnosis in the Open Learner Model

Rational The use of the OLM we can be distinguished between navigation acts (i.e. selecting the topics to investigate, browsing the model for information, etc.) and interaction acts (i.e. argumentation between the OLM and the learners about performance/behavior, etc.). The navigation acts will be addressed once the GUI of the OLM and the interface between the OLM and LeActiveMath will be better defined. This document focus mostly on the interaction acts.

Description Below is an overview of some of possible dialogue moves that the learner and the OLM could use to establish a diagnosis dialogue about the learner's beliefs and performances. Discussion and negotiation will be based mostly on judgements made by the OLM, regarding knowledge, competence and skills, emotional and motivational states of the learner. We will refer to them as *topic* of discussion.

The dialogue moves are separated in two categories: the learner's moves (LEARNER) and the OLM's moves (OLM). They will be organised in a general framework for inferential diagnosis, such as a Finite State Transition Network (FSTN), as shown in figure 7. For each of the moves, the interface between the various relevant LeActiveMath's components is described.

LEARNER: "Show me" "Show me" is the basic request a learner could use to enquire about the system's perception of his/her achievement. The focus for the enquiry could be knowledge, self-knowledge, affective state, etc. For example:

- show me what you think I know about differentiation
- show me what you think I know about my own self
- show me what you think I can do in the differentiation domain
- show me what you think I can do with my self knowledge
- show me how competent you think I am in mathematical terms
- show me how competent you think I am in terms of learning

When the OLM receives such a move, a relevant event is added to the LH (indicating that an inquiry has been made by the learner regarding topic X).

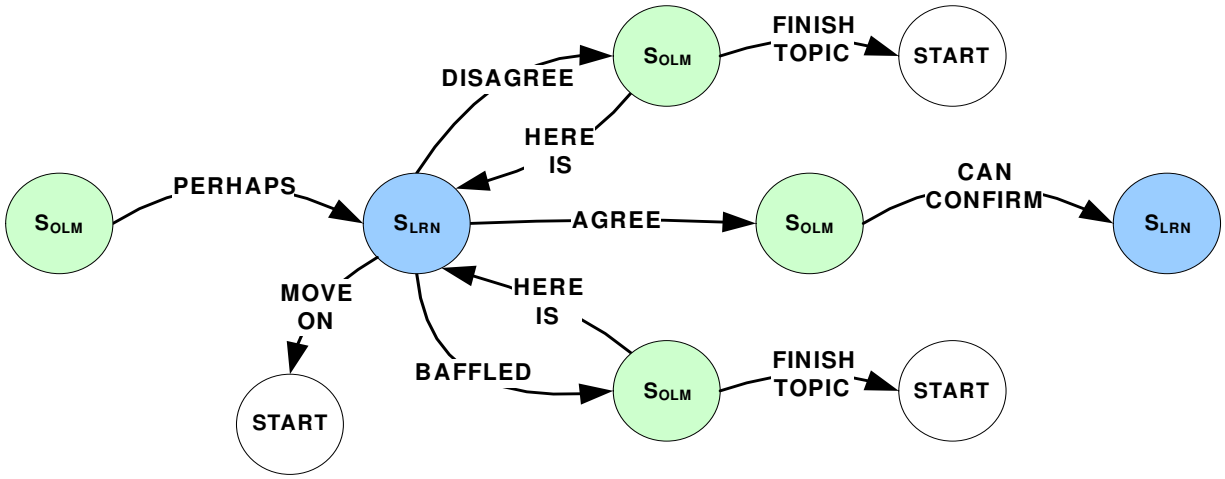

Figure 7: An example of a state transition network organising various dialogue moves for inferential diagnosis.

LEARNER: "I Agree" / I Disagree Such affirmations could be used to seed the values associated with what is being agreed with. This is primarily a response to a "Perhaps" move by the OLM. For example:

- I don't think I know the chain rule (but you seem to think I can)
- I don't think I understand the chain rule
- I don't think I can apply the chain rule
- I don't think I can improve my motivation
- Yes, I do understand the chain rule

A agree/disagree move could be moderated by the learner to express a certain amount of disagreement, such as:

- I don't think I am *very good* at mathematical modelling
- I don't think I am *very well* motivated
- Yes, I am *very* confident about my understanding of differentiation

When the OLM receives such a move, a relevant event is added to the LH (indicating that an agreement or a disagreement has been made by the learner regarding the judgement on a topic X). Indication of the learner's decision is also translated into the LM/SM.

LEARNER: "I Confirm" / "I Disconfirm" These moves follows "Can I confirm" moves by the OLM, looking for a compromise position on a particular topic. For example:

- I confirm that I think I understand the chain rule very well
- I cannot accept that I understand the chain rule quite well

When the OLM receives such a move, a relevant event is added to the LH (indicating that the learner accepts or not the compromise on the topic X). Indication of the learner's decision is also translated into the LM/SM.

LEARNER: "I am Baffled" These moves allows the learner to express his confusion regarding what is going on in the discussion with the OLM. For example:

- I do not understand what we are talking about
- I do not understand how you reach your conclusion

- I do not know where you found this information
- I do not understand why we are talking about this
- I do not understand your role

This broad category will have to be broken down at some point, as it includes moves related to both general dialogue management ("I do not understand what we are talking about") and particular judgement refutation or challenge ("I do not understand how you reach your conclusion").

When the OLM receives such a move, a relevant event is added to the LH (indicating the learner's confusion on a topic X). Indication of the learner's decision is also translated into the LM/SM.

LEARNER: "Let's Move on" These moves allows the learner to stop the current discussion on a topic and start a new line of inquiry. For example:

- I want to finish with this
- I would like to come back to this later
- I never want to revisit this

When the OLM receives such a move, a relevant event is added to the LH (indicating the learner's confusion on a topic X). Indication of the learner's decision is also translated into the LM/SM.

OLM: "Perhaps" This move is the main dialogue mean of the OLM, presenting to the learner its judgement on a topic. For example:

- Perhaps you know the chain rule very well
- Perhaps you are very good at differentiating
- Perhaps you are quite confident about using the chain rule

When asked to give its judgement on a topic, the OLM retrieves the relevant information from the LM/SM and presents it to the learner. A relevant event is also added to the LH (indicating that a judgement has been made on topic X).

OLM: "Here Is" This move is the complement of the "Perhaps" move and is only used when the learner challenge the judgement on a topic or is confused about how the OLM has made this decision. For example:

- Here is the evidence relating to your knowledge of the chain rule
- Here is the evidence I have about what you know about yourself
- Here is the evidence I have about what you can differentiate
- Here is the evidence I have about your confidence

The OLM then presents to the learner the evidence used to support this judgements. Evidences will be mostly gathered from the LH's events and made understandable by retrieving any relevant information from the LM/SM and the MBase to produce a human-readable account.

The evidences will be also internally structured to reflect the different weight that each of them may have (i.e. relevance, strength, expressivity, etc.). That structure will allows the implementation of adaptative strategies for presenting the evidences to the learners (e.g. all in a single shot, one by one starting by the most relevant, etc.). This in turn will enrich the interaction by providing the learner an opportunity to challenge accurately the decisions made by the OLM.

A relevant event is also added to the LH (indicating that evidences on topic X have been presented to the learner).

OLM: "Can I Confirm" This moves is an attempt by the OLM to conclude a line of enquiry by suggesting to the learner a commonly agreed position and is seeking a confirmation of it. For example:

- So we agree that you understand the chain rule well
- So we agree you are quite competent at mathematical modelling
- So we agree you are very good at differentiation

A relevant event is added to the LH (indicating that a proposal for agreement on topic X have been presented to the learner).

OLM: "Unravelling Confusion" This move is an answer to the "I am baffled" of the learner. For example:

- We are talking about this since knowing the chain rule is a precursor to being able to differentiate functions of the kind we are being asked to do
- I believe you are quite good at mathematical modelling because you told me so yesterday and your performance is as good as yesterday
- I think you will need to read about the chain rule or ask your teacher but we could shift to discuss ...
- I am your electronic assistant, I am here to help you increase your understanding and learn differentiation more effectively

This move is suffering the same problem as the "I am baffled", i.e. a too broad range of issues to deal with. Both moves will have to be broken down in separate categories. In any case, a relevant event is added to the LH (indicating that a suggestion has been made to clarify the confusion on topic X).

OLM: "Finish with Topic" This move is similar to the "Lets move on" of the learner, but under OLM's initiative. We might see this as a compound of "Summary" and "Where we are going". For example:

- So we are agreed that you understand the chain rule quite well - we move on/will revisit some time soon
- So we agree to differ about your competence at mathematical modelling - we move on/will revisit some time soon
- So I'd like to discuss the chain rule now and return to this later
- So I'd like to discuss your confidence now

OLM: "Can I suggest" In some circumstances (such as eliciting a disagreement or address the desire/needs of the learner), the OLM may be in a position to suggest to the learner to perform some exercises or access some content. For example:

- Since we disagree about your competence at mathematical modelling, maybe you could perform this exercise to clarify the situation
- If you want to improve your competency in mathematical modelling, I suggest you have a go with this series of exercise.

16.4 Profile

16.4.1 Configuration and Lifecycle

16.4.2 Java API

In this section the interface methods involved in the use cases (section 16.3) are given names and arguments. New methods are introduced in this section though, which consist of simple variations

of methods introduced in the use cases. When describing the methods, a combination of usual Java conventions for naming basic types and classes, creating identifiers and declaring functions is used together with Prolog conventions for describing input and output arguments to methods.

Definitions of types/classes are yet tentative, and sometimes various possibilities are considered.

Creating a learner model

- `boolean newLearnerModel( +learnerId, +domainId, +prefsAndTraits)`

  `boolean existLearnerModel( +learnerId, +domainId)`

  `boolean destroyLearnerModel( +learnerId, +domainId)`

  `learnerId` is a learner identifier, which can be either a number or a string.

  `domainId` is a domain identifier, which also can be either a number or a string.

  `prefsAndTraits` is a description of a learner's preferences and traits, probably as "learner metadata" in an XML format, to allow for great flexibility, but it can also be a Java object.

Updating a learner model A learner model is updated as a result of either XLM being notified of new events, or XLM taking the initiative (e.g. because of maintenance reasons). In the first case an interface method in XLM is needed, to be called when new events are available. On the other hand, in the latter case, XLM will be calling suitable methods of DK, hence there is no need to define an interface method for LM in this case.

- `boolean processNewEvent( +learnerId, +event)`

  `boolean processNewEvents( +learnerId, +eventList)`

  `learnerId` is a learner identifier, as in section 16.4.2.

  `event` is a description of an event, as an instance of class Event.

  `eventList` is a list of events, i.e. instances of class Event.

Exposing a learner model

- `LearnerModel getLearnerModel( +learnerId, +domainId)`
- `Face getAutApp(+learnerId, -aut, -app)`

  `learnerId` is a learner identifier, as in section 16.4.2.

  `domainId` is a domain identifier, as in section 16.4.2.

  `aut` is a numerical value calculated by the situational model which refers to the recommneded amount of autonomy that the learner should be given in the current situation.

  `app` is a numerical value calculated by the situational model which refers to the recommneded amount of approval that the learner should be given in the current situation.

The result of calling `getLearnerModel` is an object of class `LearnerModel` to be defined. At least, it will have methods for retrieving the beliefs stored in it (whose external representation is still to be defined).

The result of calling `getAutApp` is an object of the class `Face`, which returns the current values of autonomy and approval calculated by the situational model based on the relevant information

from the current Learner Model and based on the relevant student interface actions such as mouse movements and keyboard actions. Face refers to student's need for autonomy, i.e. the need to be given freedom of initiative, and his need for approval, i.e. his need to have his motivation maintained and/or boosted.

- `Belief getBelief( +learnerId, +domainId, +topicId)`

  `BeliefList getBeliefs( +learnerId, +domainId, +topicIdList)`

  `float getSummaryBelief( +learnerId, +domainId, +topicId)`

  `FloatList getSummaryBeliefs( +learnerId, +domainId, +topicIdList)`

  `learnerId` is a learner identifier, as in section 16.4.2.
  `domainId` is a domain identifier, as in section 16.4.2.
  `topicId` is a topic identifier, which will either a number or a string.
  `topicIdList` is a list of topic identifiers.

The result of calling `getBelief` is an instance of class Belief, which will be the external representation of the beliefs on a topic stored in a learner model in LM. Its content structure is still to be determined. On the other hand, the result of calling `getSummaryBelief` is a real number summarising the set of beliefs pertaining to the given `topicId`. The result of calling `getBeliefs` and `getSummaryBeliefs` is a list of beliefs and summary beliefs, respectively.

Exposing the Learner History

- `Collection getHistoryEntries( +learnerId, +inFilters, +outFilters)`

  `learnerId` is a learner identifier, as in section 16.4.2.
  `inFilters` is a list of filters that include the entries that match. Filtering could possibly be done on the interaction's attributes like date, type or the item the interaction was about.
  `outFilters` is a list of filters that exclude the entries that match.

The result of calling `getHistoryEntries` is a Collection of interactions, probably represented as events, that matched the given include filters, but not the exclude filters.

- `Collection getHistoryEntries(+pointers)`

  `pointers` is a list of pointers to interactions in the Learner History

The result of calling `getHistory` is o collection of precisely the interactions the pointers in the parameter list point to.

Inferential diagnosis in the Open Learner Model

- `void launchOLM(+learnerId, +domainID)`
  `void launchOLM(+learnerId, +domainID, +topicId)`

  `learnerId` is a learner identifier, as in section 16.4.2.
  `domainId` is a domain identifier, as in section 16.4.2.
  `topicId` is the identifier of a negotiation topic. string.

This method is called to launch the OLM of the learner `learnerId` on the domain `domainId`. Alternatively, it is possible to specify which negotiation topic `topicId` to initialise the OLM with.

- `Evidence getEvidences(+learnerId, +domainID, +topicId)`

  `learnerId` is a learner identifier, as in section 16.4.2.

  `domainId` is a domain identifier, as in section 16.4.2.

  `topicId` is an identifier of a negotiation topic.

  This method is called to retrieve and organise the evidence of the OLM judgement on a given topic `topicId`. The result of calling `getEvidences` is an object of type `Evidence` to be defined.

- `boolean postSuggestion(+learnerId, +domainID, +contentId)`
  `boolean postSuggestion(+learnerId, +domainID, +criteria)`

  `learnerId` is a learner identifier, as in section 16.4.2.

  `domainId` is a domain identifier, as in section 16.4.2.

  `contentId` is an identifier of a MBase content

  `criteria` is a list of criteria to use by the tutorial component to select a content from MBase

  Calling `postSuggestion` send a suggestion to the tutorial component that a particular activity associated with a specific piece of content should be presented to the learner as the next best choice. The content could either be specified directly (by its identifier `contentId`) or the Tutorial Component can be left to choose using some `criteria` such as difficulty, type of content, subject, etc.

16.4.3 Web-service interfaces

All methods defined in section 16.4.2 should be callable using XML-RPC, in order to facilitate distributed deployment of the Le ActiveMath components. Parameters with types other than the classes `PrefAndTraits`, `Event` and `EventList`, `Belief` and `BeliefList`, are either basic Java types or closely related classes (e.g. String), which have corresponding basic representations in XML-RPC. Parameters with types of the classes enumerated above would need to be treated differently, and there are a few alternatives:

a) they can be encoded in XML;
b) they can be converted to simpler types, build using XML-RPC types and structures;
c) they can be serialised and transferred as strings; or
d) custom handlers can be programmed to deal with them.

A reverse mechanism would be used to decode/recover/reconstruct the an identical copy of the original object.

16.4.4 Events

- `ShowMe (Attr: learnerId,topicId) [Tags: learnermodel/olm]`

  The OLM send this event to indicate that a learner `learnerId` is requiring information on a topic `topicId`.

- `Agree (Attr: learnerId,topicId) [Tags: learnermodel/olm]`

  The OLM send this event to indicate that a learner `learnerId` agrees on the OLM's judgment on a topic `topicId`.

- `Disagree (Attr: learnerId,topicId) [Tags: learnermodel/olm]`

  The OLM send this event to indicate that a learner `learnerId` disagrees on the OLM's judgment on a topic `topicId`.

- `Confirm (Attr: learnerId,topicId) [Tags: learnermodel/olm]`

  The OLM send this event to indicate that a learner `learnerId` confirms the agreement suggested by the OLM on a topic `topicId`.

- `Disconfirm (Attr: learnerId,topicId) [Tags: learnermodel/olm]`

  The OLM send this event to indicate that a learner `learnerId` disconfirms the agreement suggested by the OLM on a topic `topicId`.

- `Baffled (Attr: learnerId,topicId) [Tags: learnermodel/olm]`

  The OLM send this event to indicate that a learner `learnerId` is confused by a topic `topicId`.

- `MoveOn (Attr: learnerId,topicId) [Tags: learnermodel/olm]`

  The OLM send this event to indicate that a learner `learnerId` requires a change in the line of inquiry, away from topic `topicId`.

- `Perhaps (Attr: learnerId,topicId) [Tags: learnermodel/olm]`

  The OLM send this event to indicate that the OLM is presenting a learner `learnerId` a value judgment on a topic `topicId`.

- `HereIs (Attr: learnerId,topicId) [Tags: learnermodel/olm]`

  The OLM send this event to indicate that the OLM is presenting a learner `learnerId` some of the evidences justifying its value judgment on a topic `topicId`.

- `CanConfirm (Attr: learnerId,topicId) [Tags: learnermodel/olm]`

  The OLM send this event to indicate that the OLM is asking a learner `learnerId` to confirm or not an aggreed jugment on a topic `topicId`.

- `Unravelling (Attr: learnerId,topicId) [Tags: learnermodel/olm]`

  The OLM send this event to indicate that the OLM is presenting a learner `learnerId`

- `FinishTopic (Attr: learnerId,topicId) [Tags: learnermodel/olm]`

The OLM send this event to indicate that the OLM is suggesting a learner `learnerId` to end the current line of inquiry on a topic `topicId`.

- `Suggest (Attr: learnerId,topicId) [Tags: learnermodel/olm]`

  The OLM send this event to indicate that the OLM is suggesting a learner `learnerId` to undertake a given piece of content based on the topic `topicId`.

- `UserPropertyChange (Attr: ) [Tags: learnermodel/proper]`
  : TODO

- `UserBeliefChange (Attr: ) [Tags: learnermodel/proper]`
  : TODO

Input

16.4.5 User Events

In order to update its beliefs, the Learner Model will need to get input about the user's behaviour. This input will be sent to the xLM in the form of events, because this increases loose coupling. Thus, we describe which events should serve as input:

- Login
- Logout
- `ItemSeen`: the user has seen a given item
- `ExerciseFinished`: the user has finished an exercise
- `UserPropertyChange`: the user changed a property with the OLM
- `UserBeliefChange`: the user changed a belief in the OLM
- `ExerciseHelpRequested`: the user asked for help

17 Exercise System

Responsible for this component: George Goguadze.

17.1 Description

The Exercise System runs exercises on request via the web interface.

The Exercise System is a component solely made for the learners hence offers no API for other components. It consists of a controller and a set of services such as evaluation of mathematical formulas, and parsing of linear CAS syntax from the learner's answers.

17.2 Profile

The Exercise System is run from a specific webapp controller, and accesses other components as needed, such as the learner model and the content database. Components that need to control each step of an exercise can do so by asking it to run a specific exercise which consists just of an "interaction generator" declaration, which specifies a Java class to be instantiated that will produce the exercise steps on demand. The minimal exercise with the interaction generator declaration (a "virtual exercise") is stored in the content database, just like any other exercise.

17.2.1 Configuration and Lifecycle

Configuration is done through the exercises.* properties.

The exercise interpreter object is created on demand, when the controller receives an HTTP query for beginning an exercise, and is destroyed when requesting a different exercise.

17.2.2 Java API

None. There is no need for it since other components ask the system to run an exercise by producing an HTTP request, with the parameters in the URL.

17.2.3 Web-service interfaces

None. There is no need for it since other components ask the system to run an exercise by producing an HTTP request, with the parameters in the URL.

The integration of the dialogue-manager and of the domain-reasoner into the Exercise System are undergoing research and are out of scope of this document.

17.2.4 Events

Events emitted:
`ExerciseStarted (Attr: ) [Tags: InteractionExercise]`

`ExerciseStep (Attr: input) [Tags: InteractionExercise]`

`ExerciseFinished (Attr: successRate [0..1])) [Tags: InteractionExercise]`

`ExerciseHelpRequested (Attr: ) [Tags: InteractionExercise]`

18 Assembling Tool

Responsible for this component: Shah Jamal Alam.

18.1 Description

The Assembling Tool is running on the client. Its purpose is to let the learner collect and/or bookmarks items, such as learning material from ActiveMath, and assembles it at least in the form of table-of-contents.

18.2 Profile

18.2.1 Configuration and Lifecycle

When running within the ActiveMath, the learner invokes the assembling tool by a web-request, e.g. from a link in a menu. It will start over Java Network Launch Protocol (implemented in Java Web Start) in sandbox mode and will present to the learner an interface through which she can manage her assembled learning content.

The interface offers the learner, the facility to drag and drop html contents from the Web. These contents are stored to the learner's allocated workspace. Information about the content added to the assembling tool is entered by the learner, personal notes/logs are entered for the assembled content. While the tool is running, the learner can add/delete, edit her annotated notes about the learning material compiled by her.

The annotated notes and contents can be viewed in a browser which is lauched from the application.

With the user able to download the contents locally, the tool may later on provide features for making own copy (eg. burning a CD) to be used for offline purposes.

18.2.2 Java API

None. The Assembling Tool will live alone in its virtual machine.

18.2.3 Web-service interfaces

The Assembling Tool will not (and can not) run a web-service server.
The Assembling Tool will use the services of:

- ContentManager.`getRecBook`, ContentManager.`writeBookContent`)
- MBase (`getCommonName`, `getType`)
- and maybe the (extended) learner-model (`getBeliefs`...).

18.2.4 Events

Will listen to the following events:

- `ItemChanged`: will take care of updating the presentation of the item offered to the learner

- `BookChanged`: If an external party (e.g. the tutorial component) has changed the book we are considering.
- `UserBeliefChange`: To update a (possible) presentation of the learner's mastery aside of the item
- all notes-related events

19 Concept Mapping Tool

Responsible for this component: Philipp Kärger.

19.1 Description

The concept-mapping tool is a client component. It is invoked from within a Le ActiveMath presentation as an exercise or as a tool for personal drafting. Currently, the Java Network Launch Protocol [10] supported, for example, in the Java Web Start application, is used to to launch it.

19.2 Profile

19.2.1 Configuration and Lifecycle

In its default deployment, the concept-mapping tool will be launched as an interactive exercise. It will be launched using an application description downloaded from the Le ActiveMath server which will prove that an authenticated user is starting the application.

Using this proof, the concept-mapping tool is able to use some of the MBase xml-rpc services, and is able to send events.

19.2.2 Java API

none

19.2.3 XML-RPC API

none

19.2.4 Events Emitted

The concept-mapping tool will send two events which are shared with the exercise architecture:

- `ExerciseStarted`
- `ExerciseFinished`

Further research will determine wether more detailed events will be wished by such components as the learner-model or tutorial component.

Part III

Loosely Coupled Components

20 Introduction

Loosely coupled web-applications form a research domain which stems from the natural need to have users use different web-applications linked one another. In the E-learning domain, a good starting point could be [1].

The web, in its whole, is a big set of loosely coupled web-applications, many of which are collections of static web-pages. Loose coupling, between these can only be made by hand and suffers from update, quality-monitoring, manageability, and common-language problems.

Between a restricted and controlled set of web-applications, it is possible to achieve loose coupling with good quality results. From a user perspective, the requirements include

- *easy* integration: it should not take more than one click to go from one system to another
- homogeneity: both graphically and verbally, the user should not have to switch mind-set when switching application

Do note that these two requirements do not impose that, overall, the applications should either *feel the same*, or *be the same*. A long-term result of loose integration would be the ability for two teachers, collaborating remotely, to have one's students be taken to the activities offered by the other, and this, without programming.

The Le ActiveMath project intends to integrate two external servers as loosely coupled web-applications:

- The Siette assessment tool, developed and currently hosted in Malaga for its adaptive assessment tests.
- The exercise repository to browse collections of exercises, developed in Eindhoven.

The loose integration is going to happen at the most external level where the servers can have a parallel life but can exchange knowledge in order to let a client be *taken to* the other server. Resources will be made available on the servers and they will be readable or playable by the client if another server decides so.

For this integration to take place and to make sense several technical requirements need to be satisfied:

- single-sign-on: a learner using one of the servers should not need to authenticate itself into a new server. For this to happen, the servers have to *know* each others and trust requests they receive from another.
- learner model export or exposure: for any adaptivity to take place, servers should know enough about the learner. This can be solved either by export or by exposing the learner-model service. Note that this does not impose a centralized learner-model as each server may be asking, maintaining, and storing different information about the learner.
- a common domain-knowledge naming and possibly structure.
- discovery: it should be possible to discover activities about a domain-knowledge node.

The chapter presented describes the two servers' external to Le ActiveMath and the services they expose. We describe two general capabilities expected to be achieved by loosely-integrated components in order to reach the stated requirements.

21 Browser Delegation Scenario

We describe, in this section, a generic scenario of two servers that collaborate to delegate the browser of a user from one server to another and back for some interactions with a resource.

21.1 Players

The servers include:

- the *guide-server* is the server where our browser is before this scenario comes into play. This guide server has some reasons to wish to send the user's browser to the following server. These reasons are probably the result of some previous exchange of knowledge which we do not describe here.
- the *activity-server* which is the server that will serve the activity to the browser that it is delegated from the guide-server.

These two servers must be able to recognize the other and acknowledge that a request received comes from this server, probably by using a form of signature or registered, fixed, IP-addresses.

Users registred on the guide-server will need to exist (at least temporarily) on the activity server. For now, we shall assume the resources associated with the user will be created on the fly by the activity server if need be or the activity server will raise a fault. The activity server may need to ask further questions as first interaction with the user but such questions should not be needed anymore at any subsequent interaction.

21.2 Delegation Process

1. the guide-server notifies to the activity-server that it intends to send a browser to interact with a given resource. It provides at least the following arguments:
   - a *resource-identifier* (see later)
   - optionally, an *interaction type* that represents a *verb* describing what is expected the browser will have as interaction with the activity server. Examples include "run an exercise", or "see a piece of content"
   - a userId: this name should be the same name as the name inside the activity-server.
   - an amount of other optional information to allow the interaction to be best suited to the user. Provide a handle to the learner model may be a solution if the activity-server supports it. More realistically, providing a little set of values computed from the learner model and encoded within an ontology is probably a good idea. Group information, testifying possible priviledges would find a place here. Time-related information is also wished.
   - a URL-to-return-the-result-to which is an XML-RPC address, an http URL, see below
   - optionally, a URL-to-send-the-events-to can be provided so that events can flow between the servers while the interaction. happens

   In response to the notification received from the guide-server the activity-server should provide a URL-to-lead-the-browser-to, the guide-server now directs the browser to this URL. This URL should contain enough information so that the interaction can start right-away. Among, others, this means that this URL contains extra-*tickets* that would automatically log-in the browser on the activity-server when he first requests this URL.

The request response may contain extra information such as URL-to-send-events-to, or the URL of a tracking-document...

Faults may happen as the result of this request. Here is a suggested list, mostly inspired by HTTP protocol:

- 404: indicates that the resource-identifier cannot be associated with any available resource
- 403 or 401: indicates an authentication problem. For example, it can be sent if the wished user cannot access the indicated activity
- 307 or 301: indicates that the resource has moved.
- 500: indicates an internal error for which detailed information in the fault-message should occur.
- 501: indicates an unsupported interaction type indicated resource
- 503: indicates a temporary unavailability
- 601: indicates that creating such a user is not possible
- 602: indicates the need to create such a user first.

The guide-server, receiving such a fault should update its knowledge about the availability of the activity for the given user and should proceed to a *further step*.

2. interaction with the resource on the activity server happens. If provided, the tracking document can be polled to observe the exercise being run. Events can be exchanged in both directions, if provided. Some servers will want to react in a particular fashion to events. For example, receiving an event about the logout of a learner could be interpreted by logging-out the learner on this activity server, thus freeing resources.

3. when the interaction is finished, the activity-server should direct the browser to future interactions that the guide-server should indicate.

   It, thus, contacts the URL-to-return-the-result-to and submits an end-of-interaction. Parameters include the user and resource-identifier. It could also contain a numerical score, a floating number 0 and 1 which should describe the performance of the learner during this interaction. 0 should mean that the activity was performed with almost no success whereas 1 should mean complete success. This should be promptly answered by the guide-server which will provide a URL-to-come-back which will be sent to the browser as the next place to go to.

   Faults should only occur in exceptional cases here as there would, then, be no way else than indicating the error to the user which can, then, only *go back*.

21.3 About Resource Identifiers

As indicated earlier in this section, the loosely-coupled components need to speak the same language. At least, they need to have the same language to refer to:

- The nodes (also called topics) of the domain-knowledge which describe the elementary pieces of knowledge that the learner will learn
- The activities that can be reached using the delegation scenario described aboves

These, as well as many other things accessible on the web are generally called *resources*.

Within Le ActiveMath, we propose to use MBaseIDs for the topic-IDs. We do not (yet at least) specify the OMDoc nature on the back of these IDs. MBaseIDs are encoded as `mbase://collection/theory/name`

where each of `collection`, `theory`, and `name`, is a simple string without slash. It is expected that loosely integrated components will be able to offer activities *for* the given domain-knowledge units and that these activities will be queriable by the metadata query engine (see 14.1): the engine should at least be able to answer such a question as *give me all activities with respect to domain topic XXX* and should provide, as answers, a list of activity-IDs along with their *type*. Types can be anything but will, most probably, only be used if one of `exercise` or `text`. See section 14.1 about the metadata query engine for more.

The activity-IDs will be strings conformant to the URI specification [11]. The *namespace document* practice expressed in *The Architecture of the Web* [3] is probably a good approach to follow using these activity IDs: use existing URI schemas (such as `http`-URIs) and offer sensible information for, at least, developers about the activity resource. Such feedback as "This is a reference to a test. You can invoke it using the URL xxx." for a request for mime-type plain-text is certainly a good way to document activities.

Standardization on the available information to be offered about an activity has not been yet defined and will undergo further research in this project. Among others, we know requirement to be able extract, from the activity-ID, both an address to start the activity delegation, and an address to request the topics that this activity serves. The methods offered by the loosely integrated components documented here give hint about the possible actions about activity exchanges.

22 Assessment Tool

Responsible for this component: Enrique Machuca.

22.1 Description

SIETTE is an efficient web-based implementation of a Computer Adaptive Test Tool. The inference machine used is based on Item Response Theory. It is used as an external assessment tool by ActiveMath.

22.2 Profile

SIETTE is a web-based adaptive testing system released some years ago. It implements Computerized Adaptive Tests. In these tests the selection of the questions posed to students, the decision to finalize the test is accomplished adaptively.

Some new features are recently implemented. SIETTE is based upon a well-founded theory (Item Response Theory) and generates C.A.T.'s for grading or self-assessment. In contrast to other testing assessment mechanisms (usually use heuristic-based techniques), I.R.T. ensures that obtained student knowledge estimations do not depend on the items used in the estimation process. New features modify the adaptive behavior of Siette for the item selection, student assessment and the test finalization criteria. These criteria are based on the performance of the students while taking the tests. Furthermore SIETTE incorporates some adaptable features to the user profile, and can presents the test differently to teachers; to students that take the test for self-assessment (providing item correction, feedbacks and hints), and to those that take the test for grading

In adaptive testing, when a student takes a test, he will be administered items one by one in terms of his knowledge level current estimation. The item selected to be administered is the one which will make the student's knowledge estimation more accurate. After the student answers the item, his knowledge level is estimated taking the response into account. This process is carried out until his knowledge estimation is accurate enough.

22.2.1 Configuration and Lifecycle

None. There is no need for it since components in ActiveMath interact with the system (a web-app in an external server to ActiveMath) by producing an HTTP request, with the parameters in the URL, or using the browser delegation scenario described above.

22.2.2 Web-service interface

- `list-of-tests(topicIduser)` $\longrightarrow$list

  Returns a list of available tests. Will be used by the metadata query engine to answer queries as to which activity is offered about a topicId
  Params: topicId is just about any string that can appear as the ID of a domain-knowledge node.
  Returns: a list of test-ids.
  Shoulds: be fast

- `start-test-session(testId,user,learner-model-initial-values)` ⟶URL-to-lead-the-browser-to

  Initial step of the delegation scenario (careful that delegated authentication and server-trusting hasn't been solved in delegation yet)
  Params: a testId (maybe obtained by previous service)
  an username to identify in SIETTE knowledge database
  some optional parameters to configure the test, according to the knowledge level of learner

- `create-user(username,password)` ⟶

  Create the user learner model in SIETTE. In SIETTE, a student must provide a username and a password to access the tests. Although there are tests restricted to predefined sets of students, others can be freely accessed by simply supplying some optional personal information (e.g. name, surname, email, etc.)
  Params: an username to identify in SIETTE
  a password in SIETTE (should be the same in ActiveMath in order to make easier the delegation and authentication process)

- `get-results-of-previous-test-session(testId,user)` ⟶results

  Returns the result obtained by a student in the given test.
  Params: the testId which ActiveMath needs to obtain the assessment result the username of student who maked the test
  Returns: A list of pairs (topicId,mark) for each of the topics involved in the test

- `summarize-item-usage(itemId)` ⟶item-usage-list

  Optional "teacher-oriented" function. Returns back the results obtained by all the students that had taken this item as a part of a test.
  Params: an ID of the item which the teacher needs to evaluate, calibrate, observe...
  Returns: the distribution in discrete values (final_mark_obtained_by_student,frequency)
  Example: (0,0.12);(1,0.35);...(11,0.07) The marks are the knowledge levels stablished for the test

- `export-test(siette-domain-knowledge,available-exercises-for-each)` ⟶testId

  This is the whole test-export or test-creation scenario that was first done with SietteAsignaturaProducer which will be considered for reformulating more (much more) dynamically.
  Params:
  siette-domain-knowledge: a tree-form decomposition of the domain-knowledge (e.g. pulled from a table-of-contents or something more dynamic, e.g. a depth two tree computed on the fly from the MBase). It is not clear yet who will produce this. We should at least export each of these every-time a new content is launched.
  available-exercises-for-each: a record associating each domain-knowledge node with a list of exercises.
  Returns: a testId.

- `delete-test(testId)` ⟶

  Deletes or disables the test. If the test was used, it cannot be deleted, but disabled, in

order to keep up the students results (this affects tests, items, statistics, etc.)
Params: the ID of the test to be deleted

22.2.3 Events

None, as SIETTE is an external tool.

23 Exercise Repository

23.1 Description

The exercise repository will be a database containing interactive exercises, with interfaces for both humans and software. The interfaces allow the user to search the database and run the exercises. The exercises themselves will be described in the Mathbook XML standard [`http://www.riaca.win.tue.nl/products/mathbook/`]. Translation to and from OMDoc will be provided. It will also be possible to store, retreive and query OMDoc exercises directly (i.e. without first translating them to Mathbook), but running those exercises will not be done by the repository itself; this should be done by the ActiveMath environment that uses the repository.

23.2 Outline

23.2.1 Deployment and Configuration

The exercise repository will consist of a Java web archive (.war) which can be deployed in a servlet container such as Tomcat. This web archive contains the eXist XML database implementation [`http://exist.sourceforge.net`], the cocoon framework for XSLT processing [`http://xml.apache.org/cocoon/`], and any XSLT stylesheets, JSP tag libraries, OpenMath phrasebooks and other software that are necessary for running the exercises.

Deployment of the exercise repository should therefore be as simple as copying a single .war file into a servlet container. No further configuration should be necessary for the repository. However, some phrasebooks that will be used for the exercises might require some additional configuration (for instance, where to find the 'mathematica' executable).

23.2.2 Web-service interfaces

Searching Querying the database will be done by sending XQuery queries using XML-RPC. The eXist XML database implementation has out-of-the-box support for this, and its documentation can be found in the eXist developer's guide. Please note that this is not the method by which humans will directly query the database; for that purpose a user interface will be developed, which will be described in the user documentation.

For the convenience of the reader a description of the most important XML-RPC methods from the eXist developers guide has been copied below. For more detailed information, please consult the developer's guide itself [`http://exist.sourceforge.net/devguide.html`].

The XML-RPC method for retrieving a document from the database:

- `getDocumentAsString(StringString name, HashtableHashtable parameters)` ⟶ `String`String

  name: Path of the document to retrieve, e.g./db/exercises/exercise1.xml
  parameters: A struct containing key=value pairs to configure the output
  return value: A string containing the contents of the requested xml document

The XML-RPC method for executing an XQuery expression:

- `query(byte[]byte[] xquery, intint howmany, intint start, HashtableHashtable parameters)` ⟶ `St`

xquery: The XQuery expression
start: The position of the first item to be retreived from the result sequence
howmany: The maximum number of items to retreive
parameters: A struct containing key=value pairs to configure the output
return value: The result of the query

An example XQuery that returns all exercises authored by John Doe is:

```
//exercise[author/firstname='John' and author/surname='Doe']
```

The following XQuery returns the names of the documents that contain the results from the previous query:

```
for $i in //exercise[author/firstname='John' and author/surname='Doe']
  return document-uri($i)
```

The last example XQuery returns all OpenMath objects that contain an application of the sin function and no application of the exponentiation function:

```
//OMOBJ [
    exists(.//OMA/OMS[@cd='transc1' and @name='sin'])
    and not(exists(.//OMA/OMS[@cd='transc1' and @name='exp']))
]
```

Delegation The ActiveMath environment can redirect a user to the exercise repository, and back again. This allows you to integrate exercises into the ActiveMath content without much effort. How this works is described in chapter 21

23.2.3 Events

The sort of events that will be sent from the exercise repository to the ActiveMath environment are going to be highly exercise dependent. As a result no set of fixed events will be specified in this document. Instead the exercise authors shall be supplied with an API for sending events, very much like the `<system_message>` tag in OMDoc.

24 Appendix 1

24.1 XML-RPC summary

This description provides the basic types of XML-RPC so that we can reference them. They are mostly extracted from the XML-RPC specification [2].

XML-RPC method invocations send an HTTP POST request as an XML document and receive as response an XML document. Both the invocation and the response use the same value-types described here.

`String`
A sequence of Unicode characters

`ByteArray`
A sequence of bytes.

`int`
An four-byte integer.

`boolean`
1 or 0 (false or true)

`double`
double-precision floating point number

`date`
A date and time.

`struct`
A record associating keys to values (both expressed with other types).

`array`
A list of values (expressed with other types).

References

[1] P. Brusilovsky. Knowledgetree: A distributed architecture for adaptive e-learning. In *Proceedings of The Thirteenth International World Wide Web Conference, WWW 2004*, pages 104–113, New York, NY, May 02–05 2004. ACM Press. See http://www2.sis.pitt.edu/~peterb/papers/p641-brusilovsky.pdf.

[2] Dave Winer. XML-RPC specification, October 1999. http://www.xmlrpc.org/spec.

[3] Norman Walsh Ian acbos. Architecture of the world wide web, volume one, Dec 2004. http://www.w3.org/TR/webarch/.

[4] Alfred Kobsa. Personalized hypermedia and international privacy. *Communications of the ACM*, 5(45):64–67, 2002. URL http://www.ics.uci.edu/~kobsa/papers/2002-CACM-kobsa.pdf.

[5] EduardoÂ Pelegri-Llopart (lead). JSR-000154 Java(TM) Servlet Specification 2.3. Technical report, The Java Community Process, sep 2001. URL http://www.jcp.org/en/jsr/detail?id=53.

[6] Rafael Morales. Description of the interface of the Le ActiveMath learner model. Le ActiveMath project internal document rev. 1.2, School of Informatics, Northumbria University, October 2004. URL http://klein.activemath.org/LeAM_webView/3_ComponentsAndTools/architecture/learnermodel-interface.pdf. Draft.

[7] Rafael Morales. A proposal for Le ActiveMath architecture in the surroundings of the learner model. Le ActiveMath project internal document rev. 3.2, School of Informatics, Northumbria University, October 2004. Draft.

[8] LeActiveMath Partners. Leactivemath: Annex 1: "description of work". Technical report, The LeActiveMath Consortium, December 2003. URL http://klein.activemath.org/LeAM_webView/Shared/LeActiveMath_EU_project_Proposal_Annex1.pdf.

[9] LeActiveMath Partners. Requirement analysis. LeActiveMath Deliverable D5, The LeActiveMath Consortium, June 2004. URL http://klein.activemath.org/LeAM_webView/1_RequirementsAnalysis/final/claimsLAM.pdf.

[10] René Schmidt. Jsr00056: Java network launch protocol, May 2001. See http://www.jcp.org/aboutJava/communityprocess/first/jsr056/.

[11] L. Masinter T. Berners-Lee, R. Fielding. Ietf uniform resource identifiers (uri): Generic syntax, September 2004. See http://gbiv.com/protocols/uri/rev-2002/rfc2396bis.html.

[12] Trygve Reenskaug. The original mvc, 1978. http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html.

[13] C. Ullrich. Description of an instructional ontology and its application in web services for education. In *Proceedings of Workshop on Applications of Semantic Web Technologies for E-learning, SW-EL'04*, pages 17–23, Hiroshima, Japan, November 2004.