Deliverable number:	D2.1
Deliverable title:	Technical design and architecture

Delivery date:	2009/07/31
Submission date:	2009/07/31
Leading beneficiary:	University of Augsburg (UAU)
Status:	Version 10 (final)
Dissemination level:	PU (public)
Authors:	Bert Bredeweg (ed.), Elisabeth André, Nikolaus Bee, René Bühling, Jose Manuel Gómez-Pérez, Markus Häring, Jochem Liem, Floris Linnebank, Brenda Thanh Tu Nguyen, Michal Trna and Michael Wißner.

Project number:	231526
Project acronym:	DynaLearn
Project title:	DynaLearn - Engaging and informed tools for learning conceptual system knowledge
Starting date:	February 1st, 2009
Duration:	36 Months
Call identifier:	FP7-ICT-2007-3
Funding scheme:	Collaborative project (STREP)

Abstract

This document presents the overall design and architecture of the DynaLearn system. It integrates the different lines of technological advancements within the project DynaLearn, and details the plans to be carried out in work packages 3, 4 and 5.

The design specifies the working of the three main components, being conceptual modelling, semantic technology, and virtual characters. To a certain level of detail it also defines the subcomponents within each of these. The document includes the design of the data and knowledge exchange between the conceptual modelling environment and the arena of virtual characters, and between the modelling environment and semantic web technology. It also details the design of the overall user interface and features required for the virtual agents.

Acknowledgements

We would like to acknowledge the ideas and suggestions put forward by the partners working in WP6 and WP7, particularly the valuable contributions provided by Paulo Salles (FUB).

Document History

Version	Modification(s)	Date	Author(s)
01	Outline, draft ideas (in PowerPoint)	2009/02/17	Bredeweg
02	Moved to MSWord, processing feedback UVA DynaLearn group	2009/03/24	Bredeweg
03	Added more details, processed comments by Linnebank.	2009/04/04	Bredeweg
04a	Outline using new DynaLearn template	2009/06/24	Linnebank
04b	Input ST component received	2009/07/03	Goméz-Peréz
04c	Input VC component received	2009/07/03	André, Bühling
04d	Input CM component received	2009/07/03	Linnebank
04e	Improved input VC component received	2009/07/06	André, Bühling
5	Integrated/formatted all available text	2009/07/06	Bredeweg
6a	Update ST (2nd round)	2009/07/17	Goméz-Peréz
6b	Updated VC (2nd round)	2009/07/17	André, Bühling, Wissner
6c	Updated CM (2nd round)	2009/07/21	Linnebank, Liem
7	Integrated / formatted all available text / provided comments for further details	2009/07/24	Bredeweg
8a	Update ST (3rd round)	2009/07/30	Goméz-Peréz
8b	Updated VC (3rd round)	2009/07/30	André, Bühling, Wissner
8c	Updated CM (3rd round)	2009/07/30	Linnebank, Liem
9	Integrated / formatted all available text / provided missing parts (particularly Section 3.1 / Graphics / and Abstract)	2009/08/01	Bredeweg
10 final	Finalising	2009/08/02	Bredeweg

Contents

1. Introduction

The main objective of the DynaLearn project is to develop an interactive learning environment that allows learners to construct their conceptual system knowledge, either individually or in a collaborative setting. The workbench has three strategic characteristics:

- Accommodate the true nature of conceptual knowledge
- Be engaging by using personified agent technology
- React to the individual knowledge needs of learners

This document describes the overall architecture of the learning environment. It focuses both on the key components as such, and on the interoperability design between them.

The main components of the DynaLearn software are Conceptual Modelling (CM), Semantic Technology (ST) and Virtual Characters (VC) (Figure 1.1). The CM component (WP3) allows the learner to capture their conceptual knowledge about system behaviour, and allows for simulations based on that knowledge. The VC (WP5) should engage the learner by interacting with the learner in a knowledgeable way and provide support. The ST (WP4) allows learners to find models that are interesting content-wise, and also allows them to search and interact with learners that work on models with the same topic.

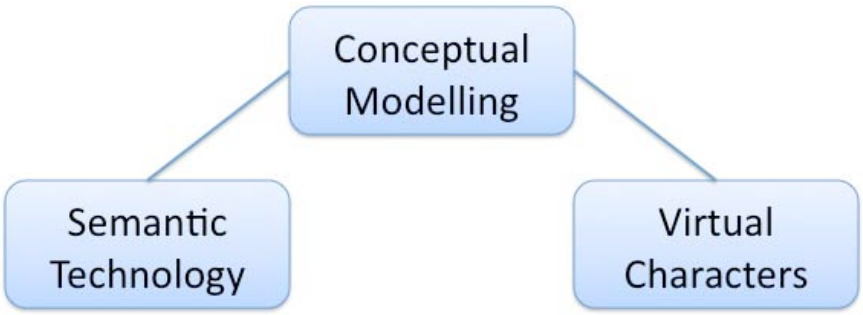

Figure 1.1: DynaLearn software components

This document first gives an overall architecture of the system and present key use-cases that describe how learners will interact with the tool (Section 2). This section also goes into a functional view of the tool, the technologies used, and describes global API's per use-case.

Section 3 describes the learning spaces, which allow learners to formalise their knowledge. It focuses on the type of knowledge learners can express, the design of the overall user interface (per learning space), and the visualisation of the inferences made in each learning space. Section 3 continues with a discussion of the question generation and diagnosis tasks, the multi-lingual issue, and the planning of the WP3 subtasks. For each use-case an API is defined that describes the knowledge that has to be communicated between the components.

The Semantic Technology section (Section 4) describes the grounding, ontology feedback and recommendation use-cases, their architecture, and the planning to develop these tasks. An API is defined that describes how the other components can interact with the ST component.

The Virtual Character section (Section 5) describes the design of the dialogue functionality for the virtual characters. It describes the use-cases that have virtual characters explain terminology, functionality and why models give certain results. Next, it describes the teachable agent, critic, model comparison, quiz and the learning companion use-cases. An architecture and API is described for each use-case.

2. Global design of the DynaLearn system

The DynaLearn software consists of three main components: Conceptual Modelling (CM), Semantic Technology (ST), and Virtual Characters (VC). This section describes the global architecture of this software taking a *use-case* perspective. The description starts an enumeration of the use-cases (Section 2.1), following the specifications in the Description of Work (DoW) [3]. Next, a high level logical view is presented (Section 2.2). This view identifies the key subcomponents in each of the three main components (CM, ST, and VC). Section 2.3 briefly enumerates a few technology related issues within the DynaLearn project. Finally, Section 2.4 provides a high level description of the communication (API) between the three main components, and its relationship with the learner (and teacher if appropriate).

2.1. Use-case view

This section enumerates and briefly summarises the DynaLearn use-cases (Table 2.1). Two actors are particularly relevant for the system, the student and the teacher (or domain expert). A typical course using the DynaLearn system may start with use-case nr. 1 where the teacher builds several target models. Whilst building these models the terms in the model will be grounded on large-scale ontologies (use-case 2). A student enrolled in the course may then start to build his own models (use-cases 1 and 2). He or she may acquire help from the basic help agent (use-case 3), receive feedback on the model being built (use-case 6) or be quizzed about the behaviour of the model (use-case 8). Other activities are teaching an agent (use-case 4), observing the model comparison dialogue (use-case 5) or interacting with a peer learner via an agent (use-case 7). These activities help directly, as well as indirectly, with the model building effort and the learning process. Note that in all student use-cases the initiative will lie with the student. The system will not initiate and enforce interactions to ensure a positive user experience.

Table 2.1: Use-cases for the DynaLearn system.

Nr	Use-case	Actor	Component	Explanation and architectural relevance
1	Build model	Student, Domain expert	CM	The CM component's primary functionality is to provide the modelling and simulation environment.
2	Ground model	Student, Domain expert	ST, (CM)	In order to connect the current model to other models in the repository it must be grounded during construction
3	Basic help	Student	VC, (ST/CM)	A VC will supply help based on input located in the ST or CM component.
4	Teachable agent	Student	VC, CM	A VC may be taught by modelling.
5	Model comparison	Student	VC, ST	Agents discuss models based on recommendations supplied by the ST component
6	Critic	Student	VC, ST, CM	Both CM and ST components will supply input for the critic agent which supports model building with content based feedback
7	Interact with peers	Student	VC	A virtual link to peer learners is established through an agent
8	Quiz	Student	VC, CM	The student is quizzed by an agent driven by questions generated by the CM component

2.2. Logical view

This section describes the DynaLearn system from the logical functional point of view. The prominent functionalities the software needs to deliver are given in Table 2.2.

Table 2.2: Prominent functionalities.

Component	Functionality	Use-case	Deliverable
CM, (ST,VC)	Communication, API's	n/a	D3.2
CM	Building and simulation workbench with use-levels	1	D3.1
ST	Model grounding system	2	D4.1
CM	Question generation system	8	D3.3
CM	Diagnosis assessment system	7	D3.4
ST	Ontology comparison system	7	D4.2
ST	Recommendation system	5	D4.3, D4.4
VC	Help agent	3	D5.3
VC	Teachable agent	4	D5.3
VC	Comparing agents	5	D5.4
VC	Critic agent	7	D5.4
VC	Virtual peer agent	6	D5.4
VC	Quiz agent	8	D5.4

An overview of the global architecture including all three components their subcomponents and primary functionality implementing these functionalities is given in Figure 2.1.

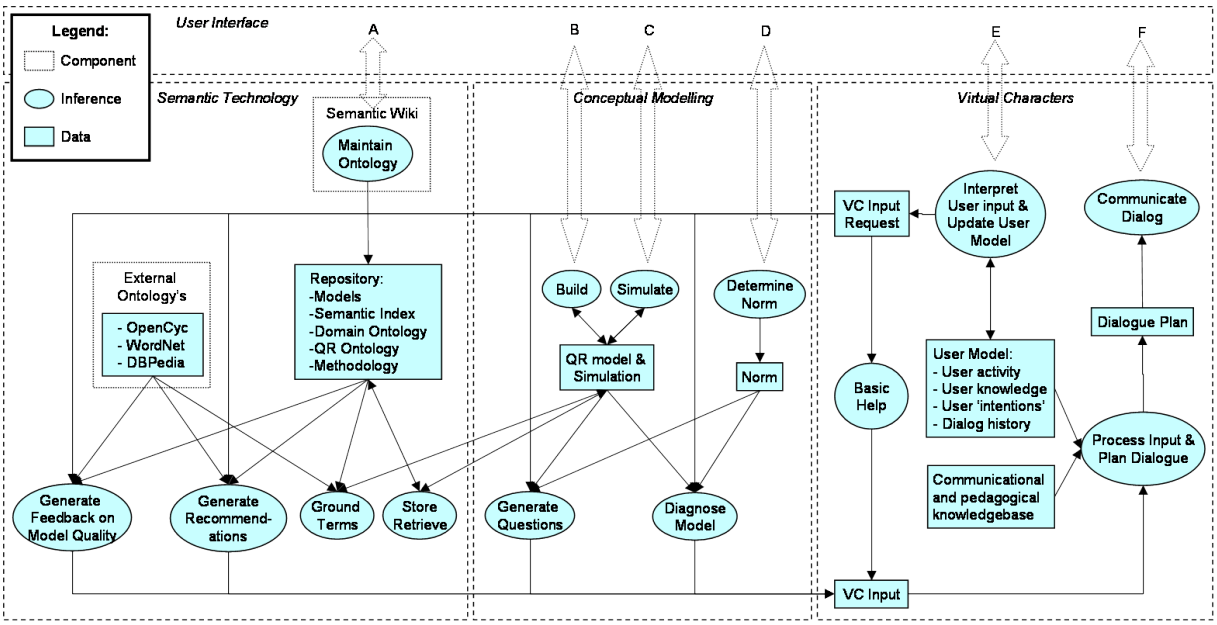

Figure 2.1: Global architecture.

Building a model, use-case 1 of Table 2.1, is implemented by interactions B and C of this figure. Grounding of terms, use-case 2, is also implemented by interaction B as grounding takes place during modelling. Also interaction A may be relevant to ground novel terms. Use-case 3, 4, 5, 6, 7 and 8 are implemented primarily by interactions E and F since they mainly consist of interactions with VCs. These will share the general VC subcomponents and differ in appearance and communicative behaviour. For the content of these interactions the VC component depends on the output of inferences from the ST and CM component. The diagnosis and

assessment inference, which provides input for use-case 6 (interacting with critic), also requires interaction D.

2.3. Technology

Several technologies will be used throughout the DynaLearn system. These include Prolog, Java, Adobe Flash, Adobe Air and Horde3D. The overall system will be platform independent regarding the OS.

Communication between the major components will be done via socket connections over TCP/IP. This technology allows a flexible configuration of components where connections can be made to components on the same machine but also to components hosted on other machines connected via the web. This functionality is expected to be especially relevant for the ST component, where the semantic repository may well be centrally located.

The CM component will act as the central hub in the system in the sense that communications between the ST and VC components will be relayed via the CM component.

2.4. API – Global component communication

To facilitate the communication between the three main components (Figure 2.2), the request-response pattern is used (Figure 2.3). That is, the first component that has the initiative sends a request to the second component, and second component responds either by sending information, or performing an action and communicating its success or failure.

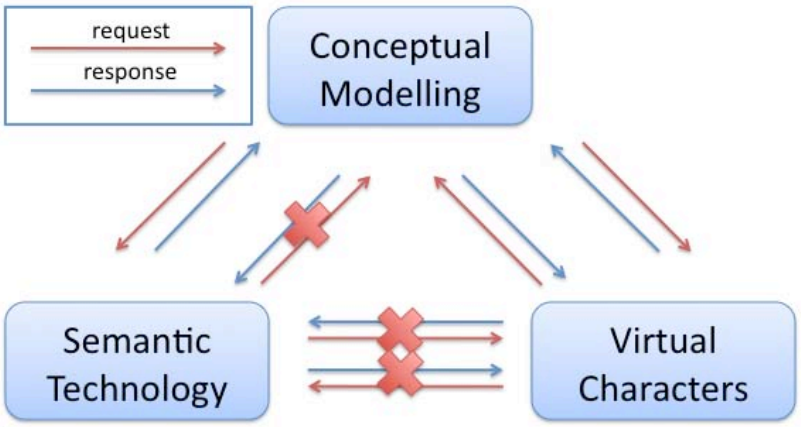

Figure 2.2: The communication between the three main components within DynaLearn.

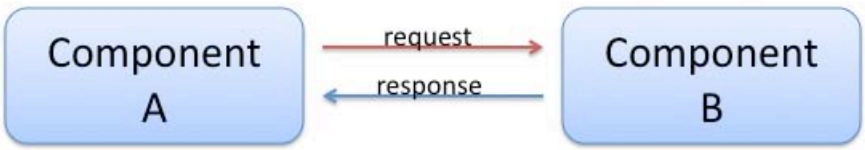

Figure 2.3: The request-response pattern.

Within the DynaLearn software, learner interaction is performed either through the CM component (when building or simulating a model) or through the VC (when interacting with a character) (Figure 2.2). As a guideline, the software should only interact with the learner when

this is requested. As such, requests are only communicated by the CM and VC components, since these components can react to learner initiatives.

The learner can start virtual characters from the CM component. This sends a request from the CM to the VC component. However, afterwards the VC is autonomous, and may take initiative to perform interactions.

To have centralised control, requests from the VC to the ST are routed through the CM component. There are several reasons. Firstly, the main model and simulation data structures are present in the CM component. These almost always need to be communicated to the ST for specific tasks. As such, it is better to redirect this communication via the CM. A second reason is to remove the need to have interactions between the VC and ST components, which would require additional formalisations and implementations.

Initially all communication interactions will be done through direct socket connections. Particularly the ST component might be transformed into a Web Service later on during the project, since the ST is running on server on the web instead of local such as the CM and VC components. A web service will have less trouble with firewalls than the direct socket connections. The ST will disconnect after each communication interaction, since it is more like a service. By contrast, the CM – VC connection will be persistent, since communication will be more frequent.

The knowledge representation language in which the communication will be done is the Web Ontology Language (OWL) [2]. OWL is part of the semantic web vision, and an established W3C recommendation. It has a large user base, and a lot of tools are being developed for its use. OWL is built upon other W3C recommendations, such as the Resource Description Format (RDF) and the Extensible Markup Language (XML).

The choice for RDF above XML is due to the more expressive (graph) representation of RDF (using *subject-predicate-object* relationships) compared to the tree representation of XML. The graph representation is more appropriate to represent the Qualitative Models used in DynaLearn. The choice to go from RDF to OWL is to be able to use OWL aware tools. Moreover, by making the representations explicit in OWL, they can also be more easily reused in different contexts.

2.4.1. Use-case 1 – Building and simulating knowledge

The details concerning the overall working of the workbench are described in Section 3. Other components, particularly the VC should be able to run a simulation.

- runSimulation

2.4.2. Use-case 2 – Model grounding

The grounding functionality is currently envisioned to work per term. That is, for each term in the QR model corresponding concepts in the external resources are sought. When a term is grounded, a term is communicated from the CM to the ST. Besides the term some of its context (e.g. its ancestor terms and children concepts) is communicated. The ST responds with a list of possible groundings that include, the names of the terms, their provenance (where does the term come from), a short description of the term, its ancestor concepts and its children concepts. Once the modeller selects the preferred grounding, the CM sends a grounding command, indicating the URI of the term in the model, and the URI('s) or the concepts in the external resources.

Learners are potentially not online all the time (e.g. while travelling with a laptop). As such, it has to be possible to ground a complete model once the user regains Internet connectivity.

The complete grounding is envisioned as using the same functionality as the grounding of individual terms. However, we will consider creating a separate call that sends the entire model so the order of term grounding can be optimized to allow for greater precision.

A special case of grounding is auto-completion. Based on term sent by the CM, the ST provides textual recommendations.

2.4.3. Use-case 3 – Basic help

Three types of basic help are envisioned in the DynaLearn approach.

2.4.3.1. Basic help – What is?

The What is? help focuses on ingredients, mainly in the model build context. Learners can choose a model ingredient in a workspace and ask what it is. The CM sends a description of the model ingredient to the VC, together with links to related model ingredients (e.g. to which it is connected, or about the class of model ingredients). The learner can choose a model ingredient proposed by the VC. This sends a new help request to the CM, which responds with a description of that particular model ingredient (and links to related model ingredients).

2.4.3.2. Basic help – How to use?

The How to use? help focuses on tasks; the activities can be performed within a particular workspace, e.g. how to delete or add an ingredient. A learner can ask for help in a specific workspace. This sends a help request to the VC. This request includes the context (i.e. the editor in which the learner is currently working. The VC utilizes a resource describing the possible functionality in that context. The VC presents the categories of actions for the specific context, and given a chosen action the possible model ingredients on which this action can be performed. Given a choice of action and a model ingredient the VC gives an explanation how this action can be performed.

2.4.3.3. Basic help – Why?

The Why? help focuses on simulation results. Within the simulation environment, the learner can choose model ingredients (such as particular magnitude and derivative values, or specific ingredients), and ask why this ingredient is present, or why a quantity has a particular value. The CM sends a template of an argument/explanation to the VC. The VC creates the verbalisation and maintains the dialogue with the learner based on that.

2.4.4. Use-case 4 – Teachable agent

The teachable agent use-case allows the learner to improve the knowledge of a virtual character. The learner does this by adapting the qualitative model that represents the current knowledge of the virtual character. At any point the learner can ask the virtual character to explain what it knows. Depending on the difficulty of the implementation, the virtual character might compete with other virtual characters, whose knowledge is based either on other learner models, or an expert model. If the learner asks the virtual character to display their current knowledge, the VC sends a request to the CM requesting specific knowledge. The CM responds back to the VC by sending back that knowledge.

2.4.5. Use-case 5 – Model comparison / Recommendation

The learner can ask the DynaLearn tool to recommend models. This is done based on a particular model (or potentially multiple models). The CM sends a recommendation request to the ST. This model should already be grounded for this functionality to work. The ST returns a list of models to the CM that includes their names, ratings, short descriptions and the reasons for their recommendation.

2.4.6. Use-case 6 – Virtual peer

The learner can ask the DynaLearn tool to find co-learners that work on the same subject. The same functionality can be used as the ('4.4.4.2 Provide recommendations'). The VC asks the ST (via the CM) what users are online and returns their contact information (e.g. Skype IDs).

2.4.7. Use-case 7– Critic / Teacher

The Critic/Teacher use-case makes use of two types of reasoning: (*i*) the quality feedback of the ST component and (*ii*) the diagnosis from the CM component.

Start Teacher / Critic Agent Request (CM → VC). A student can ask for quality feedback about his or her model, an assessment of potential issues, or a diagnosis of a specific issue encountered during modelling. Assessment thus searches for issues within a model that can be diagnosed or repaired, while diagnosis requires the learner to specify an issue specifically. The quality feedback gives feedback about the consistency and completeness compared to other models. In contrast, the assessment and diagnosis are done based on the knowledge representation and reasoning of the model. Independent of the type of reasoning used, a start critic request is sent from the CM component to the VC component, causing the virtual character to appear. The VC component returns a message indicating whether the virtual character successfully started.

2.4.7.1. Quality feedback

The learner can request feedback on the current quality of the model. This sends a feedback request from the CM to the ST. This kind of feedback requires that the grounding has already been done. The ST sends feedback back based on the golden standard, which can include the external resources, expert models, and highly rated (non-expert) models.

The feedback potentially exists of the following information: (*i*) hierarchy consistency (particularly entity, agent and assumption hierarchies), (*ii*) completeness (recommending competing processes that are not modelled, structural components that are not modelled, or quantities that could be added), (*iii*) process consistency (i.e. whether the processes are modelled in the same way), and (*iv*) whether the simulation results are comparable to the results of an expert model.

2.4.7.2. Diagnosis

Send help request (VC → CM). In both the diagnosis and the assessment use-cases a help request is sent from the CM component to the VC component. This request consists of the following content: (*I*) a help request identifier, (*ii*) a model identifier, (*iii*) a focus, for example, the entity hierarchy, a particular model fragment, or the simulation result, (*iv*) the desired result (in case of diagnosis), such as the desired simulation, or a model fragment that should fire.

Request clarification (VC → CM). In some cases the help request might be ambiguous, for example when the desired behaviour is underspecified. In such a case the CM sends a request for clarification to the VC. The VC asks the learner to for example, further specify the desired behaviour of the system (either in the form of a state graph, or by answering some yes/no questions.

Communicate discrepancies (CM → VC). As a response to the help request, the CM can communicate the discrepancies that appear in the model (based on a fault dictionary). The learner can choose one of the discrepancies that s/he wants further feedback about. The VC then sends another help request to the CM asking for a diagnosis and repair of that specific issue.

Communicate diagnosis and repair (CM → VC). As a response to the help request asking for a diagnosis, the CM returns a diagnosis and repair, which consists of: (*i*) the identifier of the help request, (*ii*) the discrepancy that was diagnosed, (*iii*) the reason that the discrepancy occurs (i.e. the diagnosis), and (*iv*) a list of possible ways the discrepancy can be repaired.

2.4.8. Use-case 8 – Quiz

In the question generation/quiz use-case, the CM and VC components interact. The CM component is responsible for the generation of the questions, while the VC component is responsible for the maintaining a dialogue history and pursuing pedagogical goals through questioning.

Starting the Quiz agent (CM → VC). As with all virtual characters, the quiz agent is activated by the learner. The learner performs an interaction with the modelling software that initiates a start quiz agent request from the CM component to the VC component. The VC component returns a message indicates whether the Virtual Character was successfully started.

Request Model/Simulation Results (Super State) (VC → CM). Since it is the responsibility of the VC component to perform dialogue history and pedagogical planning, the CM should receive a model of the knowledge that the student is supposed to learn. This model is based on the qualitative model created by the domain expert that the student is supposed to learn.

The questions generated by the Question Generator are based on the simulation results of a particular scenario. As such, the pedagogical planning should take the current knowledge about the simulation result into account. Through this end a summarized version of the simulation results (i.e. the contents of all simulation states summarized into a single state) is communicated to the VC component when requested. This ‘super state’ is formalised in OWL, and uses the same Qualitative Reasoning Vocabulary as is used to communicate the complete contents of a QR model (Section 4).

Request Question / Provide Question (VC → CM)

- VC: Request question (optional: topic)
- CM: Question + Multiple choice answer

2.4.9. Miscellaneous

This section describes a set of generic component interactions.

2.4.9.1. Semantic repository interaction

Save model (CM → ST). When the learner asks the model to be saved into the semantic repository, a save request is send from the CM to the ST. This request consists of: (*I*) the command save, (*ii*) the URI of the QR model, and (*iii*) the content of the model in OWL format [6].

Query (CM → ST). When the learner poses a query for models, the CM sends a query request to the ST. This request consists of a number of keywords. The ST responds with a list of models. For each model an abstract, authors, domain, keywords, ratings, model goals, and remarks are communicated.

Load model (CM → ST) After a query, the learner can select a model to load. The CM sends a request to the ST for a particular model (identified by its URI). The ST returns a model in OWL format, which is then loaded by the CM.

2.4.9.2. Starting VC of a specific type

Learners can ask for help from virtual characters through the interface. This sends a start character request from the CM to the VC. The VC returns a message indicating whether the character successfully started.

3. Conceptual modelling component

This section describes the architecture and design of the CM component. Section 3.1 describes the learning spaces of the workbench, which provides the primary model building and simulation environment. Section 3.2 describes the reasoning components that supply QR specific input for the VC. Section 3.3 discusses the possibilities of a multilingual implementation of the CM component. Section 3.4 contains the planning CM development.

3.1. Learning spaces

The learning spaces are organised as a set of use-levels with increasing complexity in terms of the modelling ingredients a learner can use to construct knowledge. Six use-levels have been identified and designed:

- Concept map
- Basic causal model
- Basic causal model with state-graph
- Causal differentiation
- Conditional knowledge
- Generic and reusable knowledge

The following criteria underpin the design of the use-levels and the progression between them. Most important, each level should be a self-contained interactive workspace useful for learning about system behaviour. Self-contained implies that the representational primitives available within a certain use-level form a logical subset of all the primitives available. Hence, they allow for automated reasoning on behalf of the underlying software. It also implies that from a Qualitative Systems Dynamics perspective, learners are able to create expressions that are sensible. That is, sensible in terms of the phenomena that can be perceived when observing the behaviour of systems. Moreover, what learners express using the software, will have consequences for what the software can infer. Hence, learners can be confronted with the logical consequences of their expressions, which may or may not match the observed system behaviour or their expectations thereof. Particularly in the case of such a mismatch there is ample room for interactive learning. Progression between use-levels happens by augmenting the current level with the smallest subset possible, again ensuring that the next level is self-contained. The formal context and starting point for developing the use-levels is Garp3 [1]. See also Figure A.8 (Appendix).

3.1.1. Concept map (use-level 1)

A concept map (sometimes also referred to as an entity-relation graph) is a graphical representation that consists of two primitives: nodes and arcs. Nodes reflect important concepts, while arcs show the relationships between those concepts. From a qualitative reasoning point of view, concept maps are less interesting because the do not capture dynamics properly. However, having this use-level is a logical necessity, as it is the root from which more complex knowledge representations emerge. Therefore a simple version of such a workspace is foreseen in the DynaLearn software. The essential idea of the knowledge representation at this level is

shown in Figure 3.1. The initial physical design of the workspace in shown in Figure A.1 (Appendix).

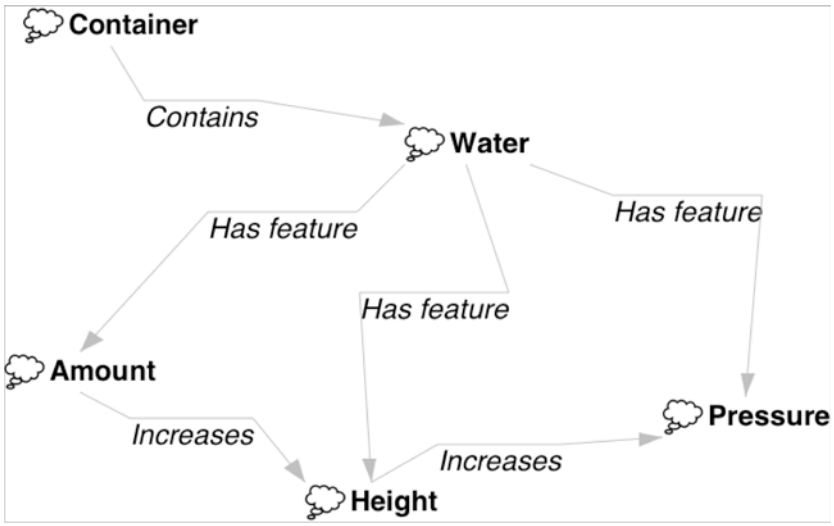

Figure 3.1: Use-level 1 – Concept map.

- Representation
  - Entity (node)
  - Configuration (arc)
- Learner actions
  - Add Entity
  - Delete Entity
  - Add Configuration
  - Delete Configuration
  - Rename Entity
  - Rename Configuration
  - Freely move icons on the screen
- Reasoning and simulation output (not applicable)
- Variations via Preferences (not applicable)
- Design issues and choices (not applicable)

3.1.2. Basic causal model (use-level 2)

The basic causal model focuses on quantities, how they change and how this change influences other quantities to change. Quantities represent behaviour and are connected to entities, the structural units in the model. Simulation at this use-level means to calculate for each quantity one of the following options (for its derivative): decrease, steady, increase, ambiguous (because of opposing influences), or unknown (because of missing information).

When constructing knowledge at this use-level the learner works in a single workspace, which effectively resembles a model fragment in Garp3. Simulation results are shown in the same model-building workspace, and have the form of derivative values generated by the software. The essential idea of the knowledge representation at this level is shown in Figure 3.2. The initial physical design of the workspace in shown in Figure A.2 (Appendix).

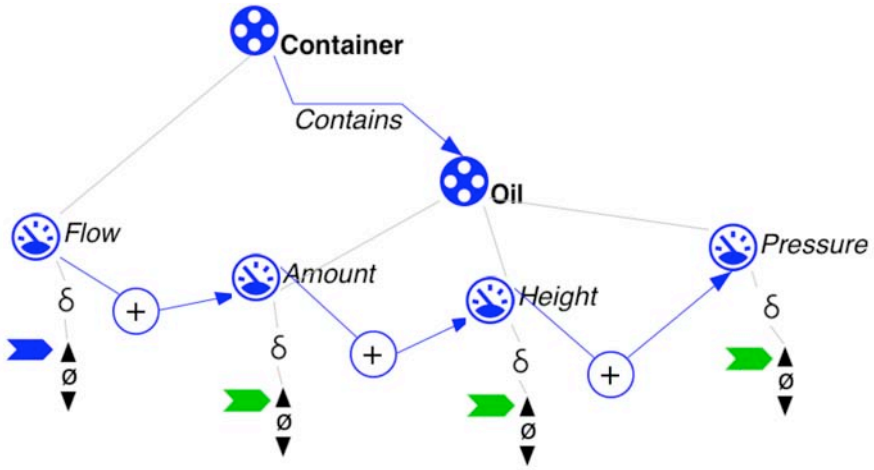

Figure 3.2: Use-level 2 – Basic causal model.

3.1.2.1. Representation

The default representational ingredients and their characteristics at this use-level are:

- Entity
  - Structural unit of the system being modelled
  - Instance level (optional)[1]
  - Type (optional)
  - One or more can be created (having at least one is essential)
- Configuration[2]
  - Structural relationship between entities
  - Holds between two entities
  - Distinction between type and instance is implicit (as in Garp3). Given the entity instance context, configurations will be seen as created at instance level.
- Quantity[3]
  - Behavioural property of an entity
  - Associated (connected) to an entity

[1] Via preference the user can select between: No structure, only instance, instance and type
[2] General for Garp3: Is there a need to refine configurations and include a hierarchy of configuration types from which instances are created.
[3] Notice that the quantity space is not included here.

  - Multiple quantities can be associated to a single entity
  - Each entity must at least have one quantity
- Derivative
  - Represents movement of a quantity: decrease, steady and increase
  - Associated to a quantity
  - Each quantity has one derivative
  - Are automatically created when a quantity is created
- Causal dependency
  - Represents a causal (and thus directed) dependency between two quantities
  - Holds between two quantities
  - Two types exist[4] and can be created: negative (+) and positive (-)[5]
- Derivative value assignment
  - Represents the current value of a derivative (magnitude and sign as in Garp3)
  - Each derivative may have one value assignment
  - Value may be unknown for the derivative of a certain quantity. In that case, no value assignment has been given.

3.1.2.2. Learner actions[6]

- The learner creates one or more entities by selecting the 'create entity option' and providing an instance name. Notice that the entity hierarchy is not included here.
- When multiple entities have been created, the learner may define configurations between pairs of entities. (When entities are not connected by a configuration the model makes a remark on the model being 'incomplete'[7].)
- The learner can assign one or more quantities to each entity. With the creation of a quantity a derivative quantity space is automatically added to the quantity (but not a quantity space for the regular magnitude).
- When multiple quantities have been created, the learner may define causal dependencies between pairs of quantities. (When none of the quantities are causally connected the model is somewhat strange and a remark could be made about that. Although the model is not incorrect as such.)
- When quantities have been created, the learner can assign values to the derivatives of each of these quantities.

[4] Requires two new icons, similar to P+ and P-, but without the P.
[5] Notice that, effectively these implement a proportionalities: P+ and P-.
[6] Notice in this documents we are not discussing re-use of previously defined models. Obviously it is possible that a learner opens an earlier created model, possibly modifies it, and runs a simulation. But this has no serious impact on the ideas described in this document.
[7] Maybe this remark should be rephrased in Garp3 by 'Incomplete structure'.

3.1.2.3. Reasoning and simulation output

The simulation result of a model at this use-level yields derivative value assignments for yet unknown values based on known values and causal dependencies (that is: propagation of changes). The results for a specific model may vary depending on the details captured in the model. Typically the following outputs will be computed (for each of the derivatives):

- Known,
- Unknown,
- Ambiguous, and
- Inconsistent.

Proper visualisation of these results is required.

3.1.2.4. Variations via preferences

The following variations on the default settings will be available

Representing entities

Notice that there is an interesting dimension regarding the accuracy of representing entities in a model, namely: None → Instance → Instance & Type → Instance & Type hierarchy[8].

- No entity (and thus also no configurations). In this case, the structural details of the model are not represented. The added value of that would be that the modelling effort becomes simpler. On the other hand, not including the structure reduces the amount of learning and insights a learner may develop. There are also indirect consequences. Not having structural details necessarily implies that quantities of similar types must be given different names, in order to distinguish them (e.g. ContainterTemperature and PipeTemperature). This blurs the model, because structure and behaviour details get mixed. This should be avoided, which is possible by including the entity in the representation.
- Instance. In this case entities are only referred to by a local (instance) name and types are not represented. This is sufficient for allowing for the use of quantity types and not having to use locally unique names for quantities. However, not having an entity type, requires mixing of type and instance names for entities (e.g. ContainerLeft and ContainerRight).
- Entity type. Next to using an instance name, the entity type can be included in the representation. The added value is that the learner is stimulated in thinking about object types. Moreover, instance names can now be refrained from also having to detail the type. On the other hand, the modelling effort becomes a bit more complex, while from a learner point of view the added value in terms of simulation results are probably negligible.
- Entity type hierarchy. In this case the entity type is embedded in a subtype hierarchy. Although there is added value in terms of learners learning more, the additional details have no impact on the simulation results. Which may make it seems superfluous from a learner point of view. Obviously, understanding hierarchies as such is an important

[8] Investigating the added value of including entities in different ways may actually be a nice research topic within one of the evaluation studies.

learning objective, and it is also a requirement to effectively construct knowledge at the higher use-levels of the DynaLearn software. However, this option will become available *only* at use-level 6, because at that level there is computational use.

3.1.2.5. Design issues and choices

The following reflections can be pointed out concerning the details discussed above.

- A clear distinction must be made between derivative values are assigned by learners and those inferred by the simulator. Learner assigned values will be shown in blue (which is the default value for that in Garp3), while inferred values will be shown in orange. Orange seems to be one of the few colours not yet used in Garp3 for a specific phenomenon. Notice that red cannot be used, due to its role already assigned in the Garp3 build context.
- Simulation results are shown in the modelling workspace. Hence, we omit the use of a state-graph and a value history[9]. This is possible because inferring only information concerning derivatives the simulation can be regarded as a 'single state situation'[10].
- Inconsistencies may occur because of assigned derivative values. This problem can be avoided by allowing learners to only assign values at the *beginning* of causal paths.
- Loops of causal dependencies may be formulated, but yield no simulation results given the current rules for calculating proportionalities (P) in Garp3. The idea is that a causal chain always includes a direct influence (I), in order to start the loop of changes. However, in this level direct influences cannot be specified, making the expression of loops impossible. *How to address this issue?*

3.1.3. Basic causal model with state-graph (use-level 3)

This use-level augments the basic causal model level with the notion of a quantity space, which can be assigned to one or more quantities. Adding this feature has a significant impact on the simulation results and necessarily introduces concepts such as state-graph, behaviour path, and value history. The essential idea of the knowledge representation at this level is shown in Figure 3.3 and 3.4. The initial physical design of the workspace in shown in Figure A.3 (Appendix).

3.1.3.1. Representation

All the ingredients from the preceding use-level (basic causal model) are included:

- Entity
- Configuration
- Quantity
- Derivative

[9] Notice that compared to the Garp3 simulate environment more details are left out. For instance, the causal model, the list of model fragments, the transition history, etc. In fact, compared to the Garp3 simulate environment we only want to visualise the value assignment for derivatives.

[10] Notice that, ambiguous results would actually require multiple states to be created: one for each possible direction of change. A unique visualisation (highlighting quantity ambiguity in such contexts) will need to be created to solve this problem.

- Causal dependency
- Derivative value assignment

New ingredients and their characteristics at this use-level are:

- Quantity space
  - Represents the magnitudes a quantity can take on
  - Associated to a quantity
  - Each quantity has one quantity space (Although, at the type level multiple may exist)
  - Must be assigned to a quantity before a quantity can be created in the workspace
  - The quantity space of a quantity can be changed in the workspace
- Quantity value assignment
  - Represents the current value of a quantity (magnitude and sign as in Garp3) and is assigned via the quantity space of the quantity.
  - Each quantity may have one value assignment
  - Values may be unknown for a certain quantity. In that case, no value assignment will be given.
- Correspondence
  - Refers to value and quantity space correspondence
  - Can be directed or bi-direction
  - Can hold between regular magnitudes, and between derivatives

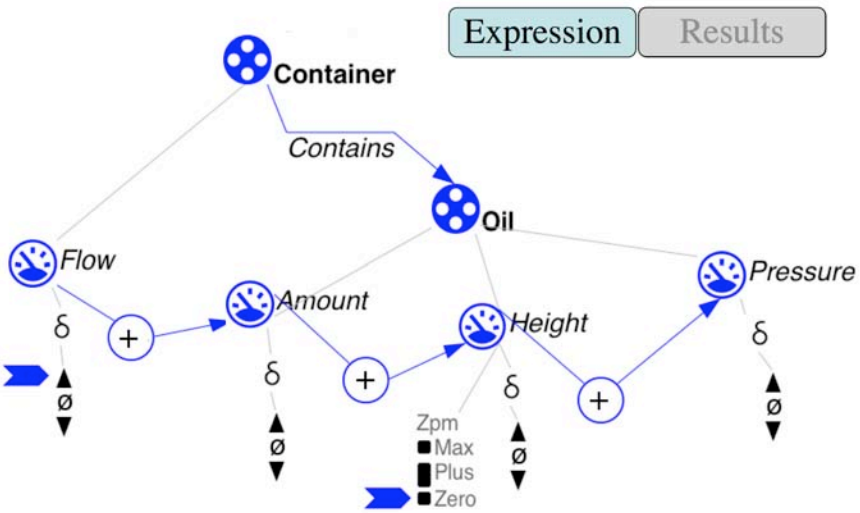

Figure 3.3: Use-level 3 – Basic causal model with state-graph (expression).

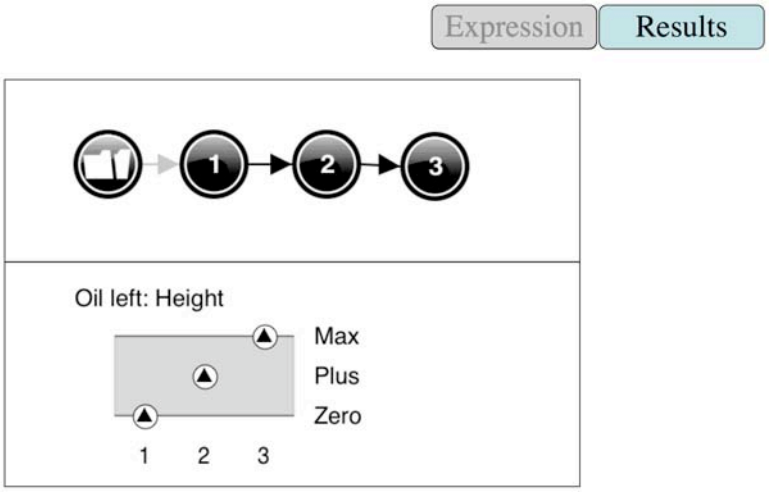

Figure 3.4: Use-level 3 – Basic causal model with state-graph (results).

3.1.3.2. Learner actions

Similar to the basic causal model the learner creates:

- Entities
- Configurations
- Quantities
- Causal dependencies
- Derivative value assignment

New at this use-level is:

- The learner needs to create one or more (generic) quantity spaces.
- For each quantity the learner may add a quantity space (see also Design issues).
- For quantities with a quantity space the learner can use idea of value assignment to specify a specific value for these quantities.

3.1.3.3. Reasoning and simulation output

Because adding the quantity space necessarily introduces advanced notions concerning the simulation results, all the simulation results will be shown in new (separate) workspaces (screens) (hence Figure 3.3 and 3.4). Essentially the simulation results are:

- Derivative value assignment (same as for the basic causal model)
- Quantity value assignment
- State
- State transition

The combination of these notions easily results in advanced state-graphs. These are difficult to comprehend without an external state-graph representation, hence the need for a workspace

showing this for a specific simulation. Similarly, the behaviour paths that exist within such a state-graph easily become difficult to comprehend without a good overview of the content details. Hence the need for a value history that shows the quantity magnitudes and quantity derivatives for a sequence of states. In summary, this use-level uses the following ingredients from the Garp3 Simulate context:

- State-graph
- State and behaviour path selection
- Value history

Notice that, when using multiple quantity spaces are used simulation results can become ambiguous, therefore the need for the state-graph, but also realise that such simulations can potentially become very big. The basic idea of ambiguity and its results is shown in Figure 3.5 and 3.6. However, such results can be very instructive from a learning point of view.

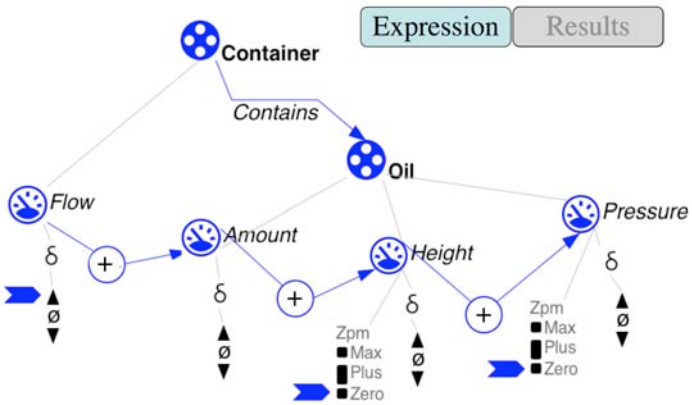

Figure 3.5: Use-level 3 – Basic causal model with state-graph (expression).

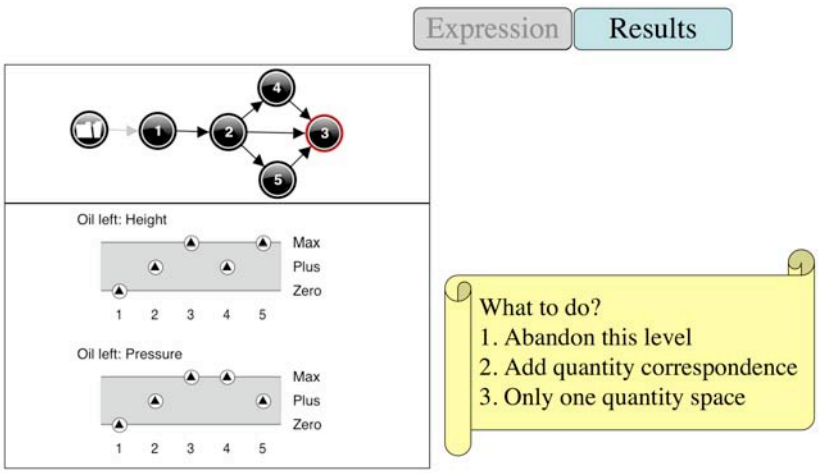

Figure 3.6: Use-level 3 – Basic causal model with state-graph (results).

3.1.3.4. Variations via preferences

The following variations on the default settings will be available:

Representing entities

Same preference options as for the use-level: basic causal model.

Exogenous quantity behaviour

For quantities at the beginning of a causal chain, the exogenous quantity behaviour option could be made available. This would include the following options:

- Derivative: Decrease, Steady, Increase, Sinusoidal, Parabola (positive), Parabola (negative), and Random.
- Magnitude: Generate all values and Constant would also be made available

Notice, there is a concern that the introduction of exogenous quantity behaviour should only happen when the notion of Agents is also present (so starting from level 4 onwards).

Default quantity spaces

Instead of learners creating quantity space a set of typical spaces can be provided for learners to choose from. For a learning point of view, creating your own quantity spaces will be insightful for a learner at some point, but being able to re-use default options may also have its merits (hence teachers can decide what the deem best in specific situations).

3.1.3.5. Design issues and choices

The following reflections can be pointed out concerning the details discussed above.

- When to create and when to assign a quantity space to a quantity? In Garp3 a quantity space is defined first, and then assigned to a quantity type. The quantity type is then used to create a quantity in a Garp3 modelling workspace (e.g. a model fragment). In other words: a quantity space is already defined and added to a quantity before this quantity is created in a modelling workspace. This is a consequence of the idea that a quantity space is part of a quantity definition, and that a type has to be defined before it can be used to create instances thereof. However, in the context of the current use-level a different order is more logical, namely after a quantity has been placed in a workspace a learner may add (and change) a quantity space (and delete it again), roughly analogous to how a quantity is added to and removed from an entity. *The approach of adding the quantity space afterwards, is the approach taken!*[11]
- To simplify matters, it could be decided that only a *single* quantity space can be defined (and assigned to one or more quantities). However, that may lead to some phenomena not being represented adequately in a shared context. Compare for instance a quantity space for substance temperature of with one for container volume. Obviously, different quantity spaces are needed to optimally capture the characteristics of each of these quantities. It is therefore better not to put restrictions on this and to allow for the creation of an unlimited number of quantity spaces.
- Single quantity space: If we allow the assignment of a quantity space for only *one* quantity in the model at the time, the simulation results could still be shown within the 'single state situation' mode. However, in such cases dynamic aspects may occur that go beyond the idea of visualising all behaviour information a single state, because the value of the quantity (with the quantity space) may change, adopting other values from its quantity space. To communicate this idea effectively, learners should see a visual representation of the state-graph and the accompanying value history. If we would stick to a single workspace, new animation ideas would have to be developed to show such

[11] Probably we assign default the quantity space 'Plus' to any quantity created. Next learners create and select other quantity spaces as needed.

changes. However, showing the simulation results in separate screens (as in Garp3) is an easier and more effective solution, both from a technical point of view as for the benefit for learning.

- Multiple quantity spaces. The 'single state situation' mode effectively becomes impossible when multiple quantity spaces are allowed. This is mainly due to potential ambiguity and each quantity space having its unique path of changing values. Hence, the cross product of these options will determine the size of the state-graph and all the possible behaviour paths through this graph. Animation, in this case becomes a cumbersome way of illustrating the wide range of possible behaviours, and declarative visualisations, such as currently available in Garp3 for state-graph and the value history, become essential.

If the approach to show all simulation results in new workspaces is fully implemented for this use-level, then the following two concerns mentioned for the preceding use-level become obsolete:

- Distinguishing assigned and inferred details using colour coding (because all inferred details are shown in other screens).
- The need to visually highlight ambiguity (because ambiguity will be shown as alternative states and behaviour paths).

The following concerns (also mentioned for the preceding use-level) remain:

- Inconsistencies may occur given assigned derivative values. This problem can be avoided by allowing learners to only assign derivative values at the *beginning* of causal paths. But this is an approach we will not take.
- Loops of causal dependencies may be formulated, but often yield no simulation results given the current rules for calculating proportionalities (P) in Garp3. The idea is that a causal chain always includes a direct influence (I), in order to start the loop of changes. However, in this level direct influences cannot be specified, making the expression of loops impossible. However, this is not an interface issue, but rather something for the diagnostic component to provide proper feedback on.

3.1.4. Causal differentiation (use-level 4)

This use-level takes all details defined for the basic causal model with state-graph and refines certain notions, particularly those related to causality. Different from the preceding use-level is that the notion of exogenous quantity behaviour is included in the default setting. Also included in the default setting is the idea of an agent. The essential idea of the knowledge representation at this level is shown in Figure 3.7, 3.8 and 3.9. The initial physical design of the workspace in shown in Figure A.4 (Appendix).

3.1.4.1. Representation

All the details from the basic causal model with state-graph are included in this use-level, hence it includes:

- Entity
- Configuration
- Quantity

- Derivative
- Causal dependency
- Derivative value assignment
- Quantity space
- Quantity value assignment

New ingredients and their characteristics at this use-level are:

- Causal dependency refinement into:
    - Influence (direct cause of change)
        - Holds between two quantities
        - Two types exist and can be created: positive (I+) and negative (I-)
        - To calculate the impact the influence, the magnitude (and sign) of the causing must be known
    - Proportionality (indirect cause of change, actually change propagation)
        - Holds between two quantities
        - Two types exist and can be created: positive (P+) and negative (P-)
        - To calculate the impact the indirect influence, the derivative of the causing must be known
- In/equality
    - Refers to relationships of the following type:
        - Binary: $A \geq B$; $A > B$; $A = B$; $A < B$ & $A \leq B$
        - Addition and subtraction: $A = B + C$ & $A = B - C$
    - May hold between
        - Quantities
        - Quantities and points in their quantity spaces
        - Between points from quantity spaces
- Correspondence
    - Refers to value and quantity space correspondence
    - Can be directed or bi-direction
    - Can hold between regular magnitudes, and between derivatives
- Agent
    - Structural unit of the system being modelled (in fact, modelling an external influence on the system, and as such ‘not part of the system’).

- Agents have quantities that are at the start of a causal chain (that is, they are not influenced themselves).
- Agent quantities may influence other quantities using proportionalities or influences
- Instance level (optional) (compare the use of Entity)
- No type (optional)
- One or more can be created
- Agents require a structural relationship (a configuration) with another structural unit (entity or agent).

- Assumption
  - Can be used (similar to agents, but causal dependencies). However, they have no impact on the reasoning. So they are kind of superfluous at this use-level…

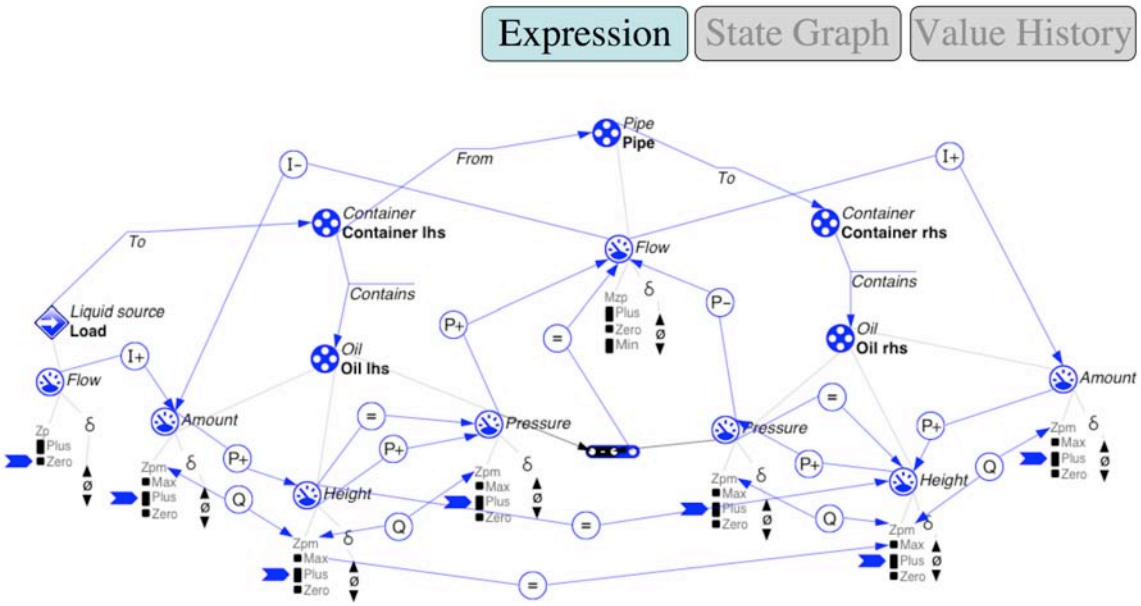

Figure 3.7: Use-level 4 – Causal differentiation (expression)

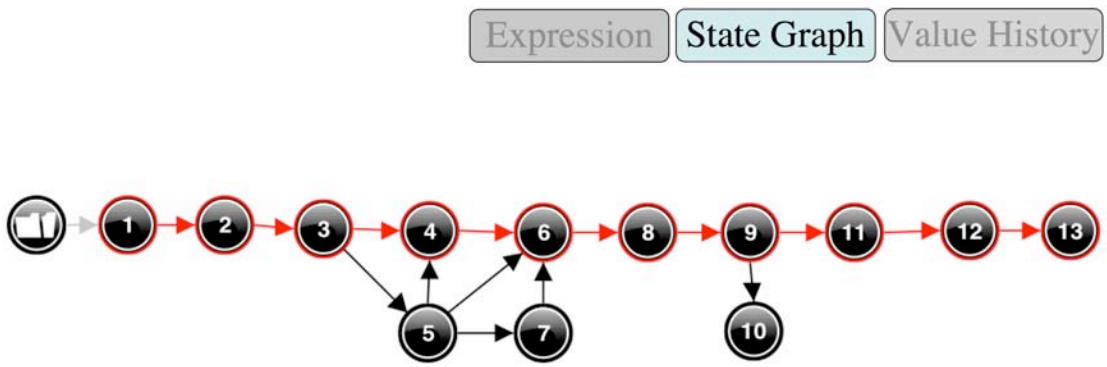

Figure 3.8: Use-level 4 – Causal differentiation (results, state-graph).

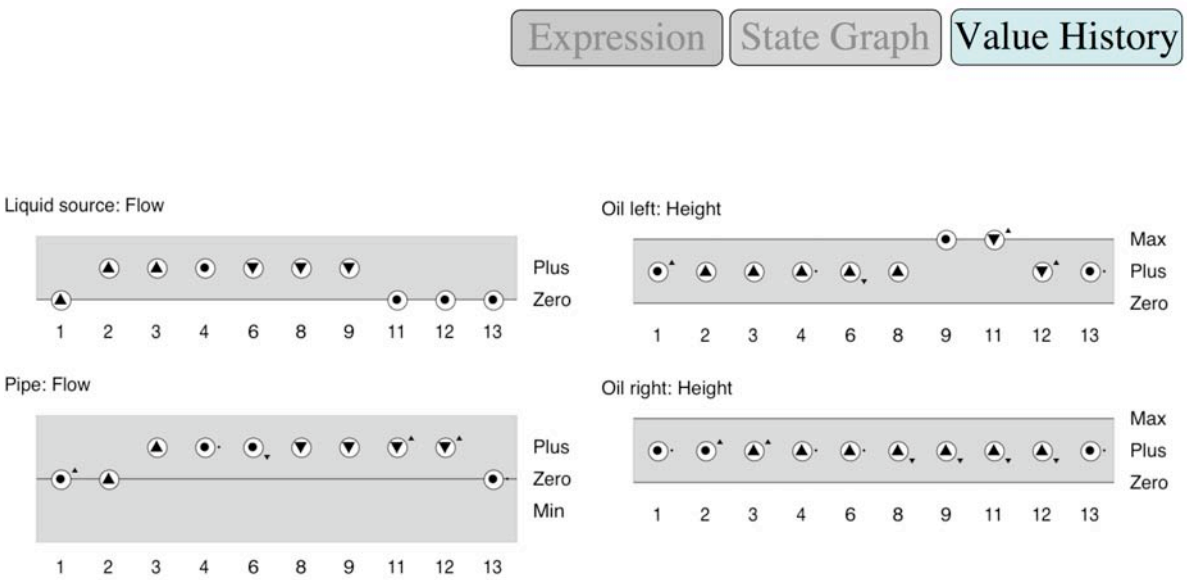

Figure 3.9: Use-level 4 – Causal differentiation (results, value history).

3.1.4.2. Learner actions

Similar to the basic causal model with state-graph the learner creates:

- Entities
- Configurations
- Quantities
- Causal dependencies
- Derivative value assignments
- Quantity spaces
- Quantity value assignments

New at this use-level is:

- The learner needs to create influences and/or proportionalities (as apposed to general causal dependencies)
- The learner may create in/equalities
- The learner may create correspondences
- The learner may create agents, which also requires additional details, including:
    - Adding configurations to complete the structure details
    - Ensuring that agents quantities situate at the start of a causal chain
    - Proper specification of initial values (magnitudes and/or derivatives)

3.1.4.3. Reasoning and simulation output

Although the internal reasoning is more complex compared to the preceding use-levels, the simulation output has the same characteristics as the basic causal model with state-graph. Hence, this use-level uses the following ingredients from the Garp3 Simulate context:

- State-graph
- State and behaviour path selection
- Value history

3.1.4.4. Variations via preferences

The following variations on the default settings will be available:

Representing entities

Same preference options as for the use-level: basic causal model with state-graph.

Exogenous quantity behaviour

The default option has now switched compared to the use-level 'basic causal model with state-graph': the exogenous quantity behaviour is included. A question is whether an option should be provided to turn this default setting off. However, it seems more appropriate to adopt the position of not making this optional, and to always include it. *We need to take a decision regarding this issue.*

Default quantity spaces

Same preference options as for the use-level: basic causal model with state-graph.

3.1.4.5. Design issues and choices

From the concerns mentioned for preceding use-levels, the following remain:

- When to create and when to assign a quantity space to a quantity? Approach at this level is still the same as for the preceding levels*: after the creation of a quantity as a separate action!*
- Using a single quantity space type? As discussed before, this is not a good idea to pursue because it severely hampers the expressive power of the learner.
- Single versus multiple quantity spaces? Although a single could be used, multiple once are almost a necessity in order to address direct influence, and affected state variables. Hence, 'single state situation' is beyond the scope of this use-level.
- Inconsistencies may occur given assigned derivative values. This problem can be avoided by allowing learners to only assign values at the *beginning* of causal paths. *But this restriction will not implemented.*
- Loops of causal dependencies may be formulated, but yield no simulation results when only proportionalities (P) are used (given the rules for calculating proportionalities in Garp3). The idea is that a causal chain always includes a direct influence (I), in order to start the loop of changes. At this use-level direct and indirect influences can be

specified, hence the problem is solved for this use-level. Feedback on this issue belongs to the diagnostic component (it is not a interface feature).

A new issue concerns the colour coding of the ingredients agent en in/equality, which are new at this use-level:

- The ingredients agent and in/equality should be coloured blue, because they are user-defined and always true

3.1.5. Conditional knowledge (use-level 5)

This use-level is a kind of refinement of the causal differentiation level. All representation details apply as they do for this preceding level. The main difference is the possibility to specify conditions under which specific set of details are true. In the preceding use-levels all expressions done by a learner are always true. However, some facts (e.g. an evaporation process) only happen when certain conditions are satisfied. The use-level conditional knowledge addresses this. The essential idea of the knowledge representation at this level is shown in Figure 3.10, 3.11, 3.12, and 3.13. The initial physical design of the workspace in shown in Figure A.5 (Appendix).

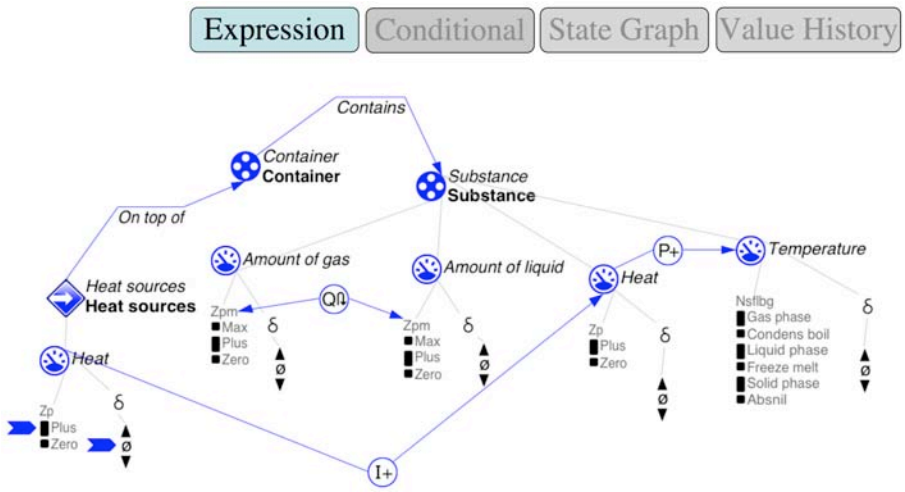

Figure 3.10: Use-level 5 – Conditional knowledge (expression).

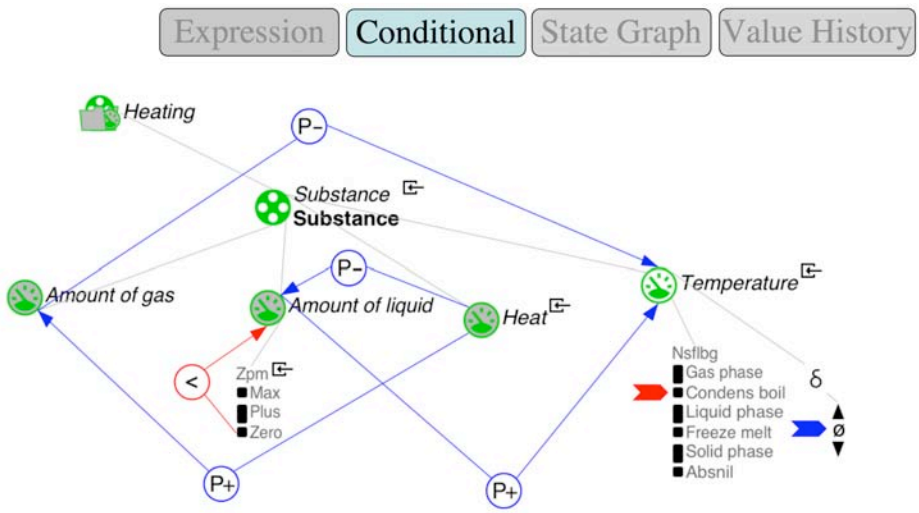

Figure 3.11: Use-level 5 – Conditional knowledge (conditional knowledge).

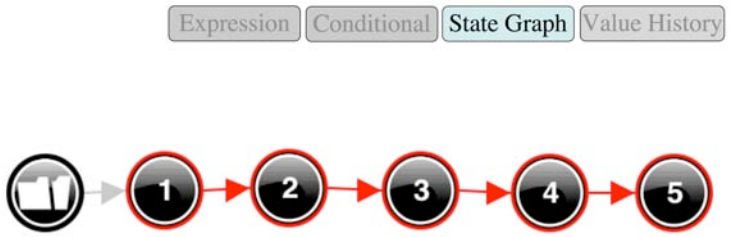

Figure 3.12: Use-level 5 – Conditional knowledge (results, state-graph).

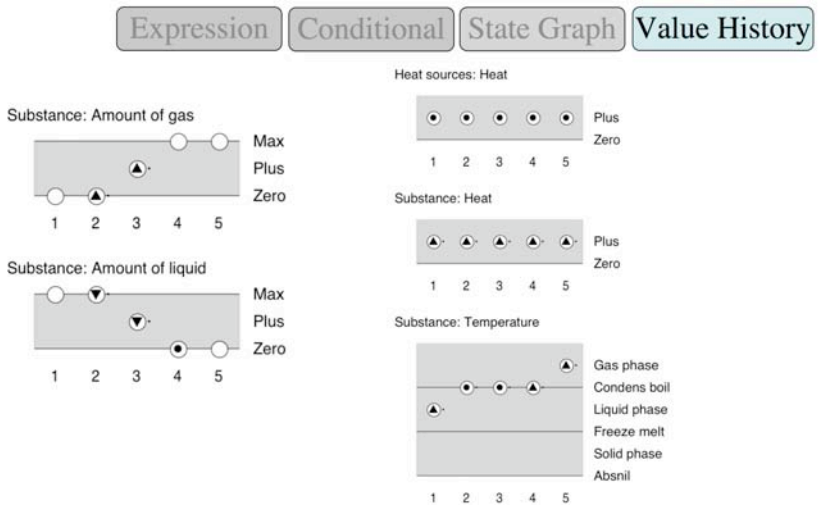

Figure 3.13: Use-level 5 – Conditional knowledge (results, value history).

Compared to the preceding use-level, the following issues require special attention:

- Representation
  - Learners can now create multiple workspaces (probably as a set of 'tabs'). The 'first' space captures facts that are always true. All succeeding spaces include the details from this first space and allow the learner to add more. When adding more ingredients, the learner has to specify for each ingredient whether it is a condition (or a consequence).
  - The following ingredients can be used as condition:
    - Entity
    - Configuration
    - Quantity
    - Derivative value assignment
    - Quantity value assignment
    - In/equality
    - Agent
- Learner actions
  - Large similar as the preceding use-level: causal differentiation.

  - New is that learners can create multiple workspaces and that they (except for within the first space) have to detail condition versus consequence for each ingredient.

- Reasoning and simulation output
  - Same as for the preceding use-level: causal differentiation.
- Variations via Preferences
  - Same as for the preceding use-level (Causal differentiation).
  - If for representing entities (and agents) the format 'Instance & Type hierarchy' is used. The entities in the 'conditional' workspace could be refined[12].
- Design issues and choices
  - Conditions will be coloured red (as in Garp3). Consequences will be coloured blue.

3.1.6. Generic and reusable knowledge (use-level 6)

This use-level reflects Garp3 in its current status. The main difference with the other use-levels is the focus on 're-usable' knowledge. That is, to capture essential details in a context independent manor as much as possible, or otherwise to explicate the conditions under which the knowledge is applicable. At this level the notion of types and hierarchy become important. Also the idea of creating ones own model by re-using (partial) solutions stored in a repository is now viable.

This document will not explain Garp3 in details, but see [1] (Figure A.7 gives on impression of the physical design):

- Representation (see Garp3)
- Learner actions (see Garp3)
- Reasoning and simulation output (see Garp3)
- Variations via Preferences (see Garp3)
- Design issues and choices
  - The operation and window manipulation will have to be given the same look-and-feel as the preceding use-levels (main point: move towards an integrated work spaces using less windows).

3.1.7. Garp3 technical and interface overall issues

This section discussed issues related to the Garp3 knowledge representation and reasoning that require a solution in order for the engine to properly work. In general, the issue is that at a certain use-level some Garp3 essential representation details are left out of the arsenal available to users. Not having such representational features may cause the Garp3 engine to produce error-messages, and possibly to fail to deliver proper simulation results. To cope with this issue, the basic idea is to have the software automatically include the missing details,

[12] This is currently not possible in Garp3 (although a 'work around is available').

without the user noticing this. As such these details should be hidden from the user while working at this use-level. The following issues will have to be considered:

- No type and/or instance entity: Add hidden type and/or instance entity to assign quantities to.
- No quantity space for a quantity: separate adding a quantity space from the quantity definition, and assign default quantity space 'plus' to quantities.
- Single workspace: add scenario and at least one model fragment, and present simulations results in the model building workspace. Store input ingredients in the scenario (e.g. a value assignment), and model ingredients in the model fragment (e.g. a quantity).
- Automatically created model fragments needs a type. One option is to always take Agent, because that allows for all ingredients to be included. A second option is to automatically select the type that actually fits the contents, e.g. when no influences and agents are defined the fragments can be of type static. A third option is to create a new type especially for this case (suggestion for type name: expression). Probably the latter option is best, because it allows all components to recognise that they are dealing with a non-regular model fragment.
- In order for components to know how to visualise and reason with a GARP3 model, each model will be given a use-level identifier. The interaction with the data can then be adjusted by a component accordingly.

3.2. Reasoning components

The main components of the CM subsystem are shown in Figure 3.14. Three of these are reasoning components: The qualitative reasoning simulation engine, which is taken from Garp3, the question generation and answering component, and the diagnosis and assessment component. As shown, the user interface accesses the model building and the simulation environments. The build component supplies input for the simulation engine and the diagnosis component. The simulation is input for the question generator and the diagnosis component. The following sections describe the latter two new reasoning components in more detail.

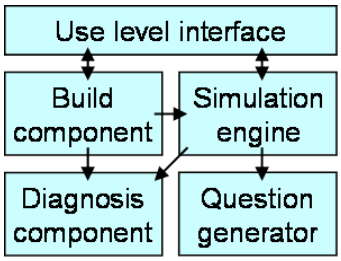

Figure 3.14: Components of the CM subsystem.

3.2.1. Question generation and answering

The question generator component is based on previous work on the QUAGS question generator [5]. This component will feed the quiz agent with questions about the simulation. The form of the quiz will be multiple-choice so the question generator will need to generate distractor items as well as correct answers for the user to choose from. The question generator will be domain independent and will make use of the model to construct questions with rich content. The

distractor items will vary in their plausibility, ranging from content that is not actually found in the question context, to content that can be found in the question context but which has a (slightly) different relation to the question.

3.2.1.1. Components

The architecture of the question generator is shown in Figure 3.15. It shows the components involved in the question selection and it is described in this section. Firstly, the *Build* and *Simulate* inferences interact with the learner (not shown) and produce a model and simulation. Secondly, the *Generate Questions* inference takes a question request as input, which may contain several constraints concerning the desired focus of the questions. This focus may be generated from the student model present in the VC component. To build a student model, information about the model and simulation is needed and this information is supplied by upon request. Further input for the actual question generation component is the model and simulation, which provide the content to base the questions on, and a set of question templates and criteria. These selection and ordering criteria are needed to trim the amount of generated questions and select the most relevant ones. Without such selection the amount of questions can be very large. A simulation with 4 quantities, 4 states and 4 causal relations for example will spawn 448 questions[13] in the QUAGS question generator. A number that is clearly not feasible to communicate to the user so a pre-selection of the most useful questions must be made.

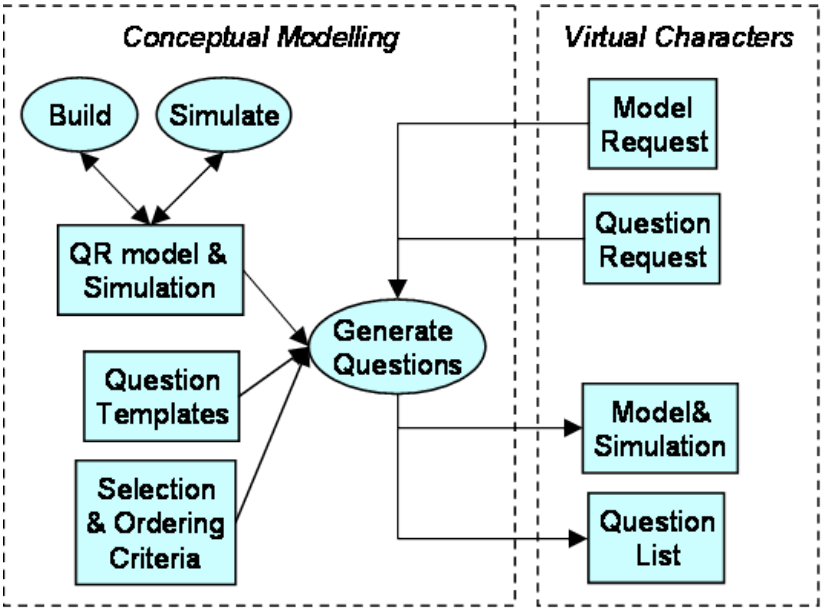

Figure 3.15: Architectural view on question generation.

3.2.1.2. API

- Get Model
  - Method name: **getModel**
  - Input parameters: Model URI or 'current'

[13] In the Quags [5] generator the total number of questions equals $8QS + 8RS + 3SQ^2$ with Q, R and S being the amount of quantities, the number of causal relations and the number of states respectively.

    - Output parameters: Model in OWL format
    - Description: Returns the model in OWL format.
- Get Simulation
    - Method name: **getSimulation**
    - Input parameters: none
    - Output parameters: SuperState summary of the simulation
    - Description: Return summary of the simulation in OWL format
- Get Questions
    - Method name: **getQuestions**
    - Input parameters: QuestionRequest
    - Output parameters: QuestionList
    - Description: Generates a set of questions based on a question request.

The QuestionRequest has the following features:

- requestID: uniquely identifies the request for questions.
- States: identifies the states considered in the question generation.
- Perspective: Consider the entire simulation or the causal model.
- Concept: Indicate which model ingredients to consider.
- Behaviour: Consider only dominant causal relationships, or also submissive ones.
- InfoState: Consider the current state or all other states.
- AnswerMethod: Difficulty of reasoning needed to find the answer (ranging from location of the answer mentioned in the question, to the answer requiring assembling pluriform information).
- SystemScope: entities and quantities about which questions can be asked.
- SubjectQuestions: quantities of which at least one *must* be involved in each question.
- NonSubjectQuantities: Quantities about which no questions may be asked.
- QuestionType: Keyword identifying the questions template used.

The QuestionList consist of multiple Questions that consist of:

- QuestionID: A unique ID identifying the question.
- RequestID: The RequestID that was in the QuestionRequest.
- QuestionType: The keyword identifying the questions template used.
- QuestionStructure: The number of answer slots, single/multiple answers, etc.

- FormalQuestionContent: Formal representation of the question including states, ingredients (quantities, values, etc),and conditions.
- FormalAnswerContent: Formal representation of the answers (including whether they are correct or not)
- VerbalizedOutputText: Question in the form of natural language,
- VerbalizedAnswer: Answer in the form of natural language

3.2.2. Diagnosis and assessment

Figure 3.16 shows the components involved in the diagnostic task. This task takes input from the VC and ST components and supplies the former with input for constructive dialogues with the learner. The following subsections will discuss the individual components, API and planning for this system.

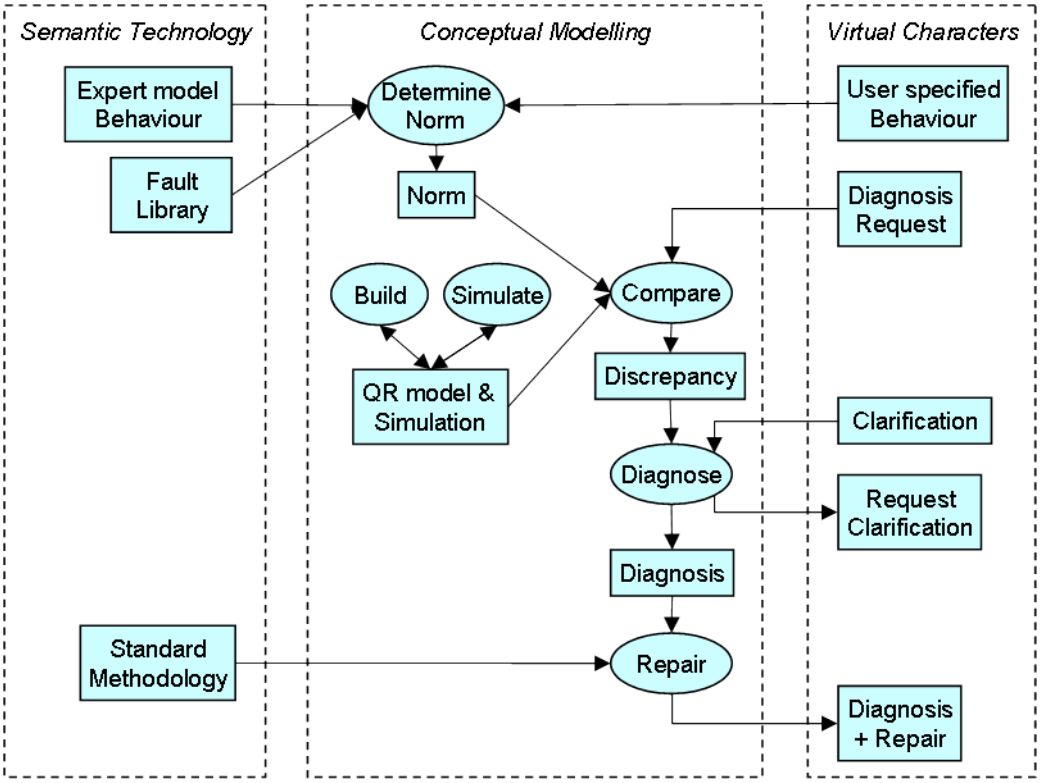

Figure 3.16: Architecture for diagnosis and assessment.

3.2.2.1. Components

The architecture shown in Figure 3.16 is described here. The first input for the diagnosis and assessment system is a diagnosis request from the VC component. This request is initiated by the learner and contains an indication problem. This indication may come in the form of (1) user expectations that are not met, (2) a need for peer or expert model comparison, or (3) a request for a general check of the model.

In the first case expected behaviour is not occurring or unwanted behaviour is occurring. So this user specified behaviour is input for the Determine Norm inference of Figure 3.16 and results in a norm that can be compared with the actual model and its behaviour. The resulting

discrepancy leads to a diagnosis, which in turn leads to a repair. The last step may involve a standard methodology to guide the repair process, as the generation of a repair may be a difficult task. Output of the system is a diagnosis and repair proposition, which is communicated to the learner via the VC component. In case the input is insufficient for a proper diagnosis a request for clarification can be sent to the VC component, which results in information elicitation and a learner clarification coming back. Hereby, an interactive diagnostic process is established.

In the second case the norm results from a peer or expert model in the repository in the ST component. The rest of the diagnostic process does not differ from the first case, but the interactivity of the process is expected to be much lower.

In the third case the norm is a fault library containing (anti-)patterns against which the model and/or the simulation is/are screened. In this case, the rest of the diagnostic process will be much reduced, as the fault library norm will also provide a diagnosis and possible repair.

3.2.2.2. API

- GetDiscrepancies
  - Method name: **getDiscrepancies**
  - Input parameters: model ID
  - Output parameters: discrepancy list
  - Description: Determine the discrepancies in a model based on a golden model.
- PerformDiagnosis
  - Method name: **performDiagnosis**
  - Input parameters: Help Request ID, Model Identifier, Focus (discrepancy), Desired result
  - Output parameters: Cause, Repairs, or a request for clarification
  - Description: Determine the cause of a discrepancy and the possible repairs.
- Diagnosis
  - Method name: **getModel**
  - Input parameters: Help Request ID, Model Identifier, Focus (discrepancy), Desired result
  - Output parameters: Cause, Repairs
  - Description: Determine the cause of a discrepancy and the possible repairs

3.3. Multilingualism

The Conceptual Modelling component will be able to provide support for a multilingual approach without major changes. The qualitative reasoning engine is domain independent and therefore language independent and functionality is available to save a model in multiple languages. There are only two small concerns. Firstly, the tool-tip texts are in English and this

would need work to be translated. For building a model this is not a crucial feature however. If it will be decided that multilingual tool-tip texts are needed then a flexible implementation with separate language files must be realised. Secondly Garp3 supports a subset of the ASCII character set, initially stemming from the limitations of the Prolog system used. Therefore the use of exotic characters needed in some languages is currently not supported. Solving this issue is technically possible, but requires effort.

3.4. Planning conceptual modelling development

This section provides the plan for creating the CM component. Each subsystem will be implemented in an iterative process. All phases will create a working version of the system. The first iterations will be more elementary and focus on research and proof of concept. The work breakdown structure and planning is given in Figure 3.17. The list of tasks in this planning ordered by deadline and including descriptions is given in Table 3.1.

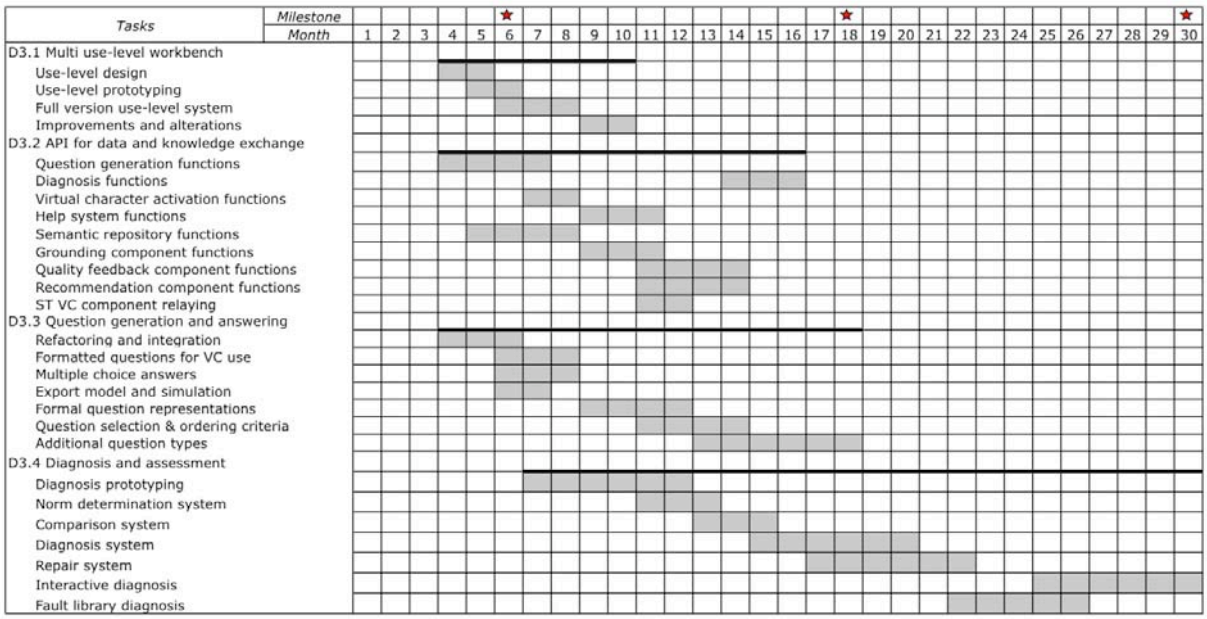

Figure 3.17: Planning of tasks for Conceptual Modelling component.

As can be seen in Figure 3.17, the CM component has four associated development areas: use-level workbench, component communication via API's, question generation and answering, and diagnosis and assessment. These are related to the four deliverables D3.1-D3.4 respectively.

The use-level workbench has four main deadlines: the interface design is due in month 5, the prototyping efforts must be concluded by month 6, a complete working system is due in month 8 (first half year meeting), and the final version is due in month 10. The component communication schedule is synchronised with the planning of the components related to each set of functions.

The question generation schedule has two main deadlines: a working prototype must be available by month 8 (first half year meeting). This version will include the integration of the question generator using natural language questions and the implementation of all necessary API functions and formats. The second version will supply formal question templates such that the VC can create the verbalisation of the questions while maintaining a proper dialogue. Also it will include a bigger set of possible questions. This version must be done by month 18 (2nd project milestone).

Each of the development cycles for the diagnosis and assessment subsystem should include a full system communicating results to the VC component. Earlier iterations will focus on simple learner expectation-based diagnosis. Later iterations will include expert model-based diagnosis, interactivity and fault library diagnosis. There are five main deadlines. First prototyping efforts to guide the development must be concluded by month 12. A first version focussing on the norm determination and comparison tasks must be done by month 15. A second version focussing on diagnosis and repair must be don by month 22. A version based on fault library diagnosis must be done by month 26 and the final version including user interaction is due in month 30.

Table 3.1: Task list for the CM component ordered by intended deadline.

Task	Due	Description	Part of
Use-level design	M5	The use-level interface design has to take into account pedagogical aspects as well as QR related constraints.	D3.1
Use-level prototyping	M6	Initial R&D on the use-level workbench implementation.	D3.1
Refactoring and integration	M6	The QUAGS question generator needs to be updated to work with the current Garp3 model format and some refactoring needs to be done before this proof of concept implementation can be integrated. in the DynaLearn software.	D3.3
Question generation functions	M7	This task covers the API functions related to the quiz use-case.	D3.2, D3.3
Export model and simulation	M7	The model and simulation must be communicated to the VC component in an appropriate format. This format needs to be defined and implemented.	D3.3
Full version use-level system	M8	The use-level workbench in a complete working version.	D3.1
Virtual character activation functions	M8	This task covers the API functions related to the all VC use-cases.	D3.2, D3.3, D3.4
Semantic repository functions	M8	This task covers the API functions related to the semantic repository (ST).	D3.2, D4.1
Formatted questions for VC use	M8	The questions and answers need to be communicated to the VC component in a way that is useable for the quiz agent.	D3.3
Multiple choice answers	M8	Distractor items for the multiple-choice quiz must be generated based on model content. An assessment must be done of the functionality of the answer set of each question type.	D3.3
D3.1 Multi use-level workbench	M10	Top level task	D3.1
Improvements and alterations	M10	Following up on the initial full version of the use-level workbench a final version will be implemented.	D3.1
Help system functions	M11	This task covers the API functions related to the basic help use-case.	D3.2, D5.3
Grounding component functions	M11	This task covers the API functions related to the grounding use-case.	D3.2, D4.1
Diagnosis prototyping	M12	Initial R&D on the diagnosis and assessment task.	D3.4
ST VC component	M12	This task involves relaying any information between the VC and ST component.	D3.2

relaying			
Formal question representations	M12	Questions are now generated in natural language (English). To make a multilingual implementation and a more flexible dialogue possible a machine understandable format must be designed and implemented.	D3.3
Norm determination system	M13	The norm determination is a pre-processing step to translate learner expectations or an expert model into a norm to which the current model can be compared. This subtask also involves the design of a format to specify expectations and a format for the norm.	D3.4
Quality feedback component functions	M14	This task covers the API functions related to the quality feedback subsystem (ST)	D3.2, D4.2
Recommendation component functions	M14	This task covers the API functions related to the recommendation subsystem (ST)	D3.2, D4.3, D4.4
Question selection & ordering criteria	M14	The current heuristic criteria for question selection and ordering must be extended if new question types are added. Also these criteria could be extracted and implemented in the VC dialogue planner where the actual question selection takes place.	D3.3
Comparison system	M15	The comparison system will compare two simulations or parts of these. This subtask will therefore include the design of a comparison algorithm making the appropriate generalisations and a mechanism for determining the appropriate scope of the comparison.	D3.4
D3.2 API for data and knowledge exchange	M16	Top level task	D3.2
Diagnosis functions	M16	This task covers the API functions related to the quiz use-case	D3.2, D3.4
D3.3 Question generation and answering	M18	Top level task	D3.3
Additional question types	M18	The current set of questions can be extended with new question types about concept such as: the structural hierarchy of the model, model fragments that may fire or not, and reasons why certain events did not happen (transitions failed).	D3.3
Diagnosis system	M19	The diagnosis system will interpret the discrepancies found by the comparison system.	D3.4
Repair system	M22	The repair system will be closely linked to the diagnosis system and generates the way to alleviate the problem initially pointed out.	D3.4
Fault library diagnosis	M26	The fault library will be assembled using case studies of modelling efforts. (Anti-)patterns may be matched on the model as well as on the simulation results.	D3.4
D3.4 Diagnosis and assessment	M30	Top level task	D3.4
Interactive diagnosis	M30	Interactive diagnosis is a part of the diagnosis system. It will resolve ambiguous diagnostic results. This task includes the generation of questions that can be posed to the learner to acquire clarifications.	D3.4

4. Semantic technology component

This section describes the architecture of the Semantic Technology (ST) component in the context of the overall DynaLearn infrastructure. We explore the main functionalities to be provided by the ST and relate them with the rest of the engineering components of DynaLearn. For this purpose, we bootstrap from the three main use-cases of this technology in the project:

- Grounding of terms and the QR models they belong to in the DynaLearn vocabulary
- Ontology-based feedback on the quality of the models produced through the modelling environment
- Recommendation of models and model fragments according to their relevance for the modeller and the properties of the model under development

From these three use-cases in this document we will focus primarily on the first two, whose requirements are more immediate and well defined at this early stage. Nevertheless, we will also provide ground for future development in the recommendation front.

Section 4.1 provides a description of such use-cases, introducing the components involved and the required functionalities. Section 4.2 presents the architecture containing such building blocks and describes the API to the functionalities exposed by the different units of the DynaLearn ST component. Finally, Section 4.3 describes a development plan, including intermediate milestones with internal software deliveries.

4.1. Use-case perspective

This section describes the use-cases Grounding (Section 4.1.1), Ontology-based quality feedback (Section 4.1.2), and Recommendations (Section 4.1.3).

4.1.1. Grounding

It is necessary that terms produced by users (learners and teachers), while constructing expressions using the CM component, get grounded in well-formed and established vocabularies. This serves a threefold purpose: (i) it ensures that the terms used and, by extension, the models they belong to, are correct both lexically and semantically, (ii) by providing a common vocabulary to the models created by different authors, we ensure that such models are interoperable at a terminological level, and (iii) we provide the means for the vocabulary to be dynamically updated through the terms added by subsequent models.

Two types of grounding are foreseen, which respond to two different levels of granularity (term and model grounding). Section 4.1.1 describes the idea of term grounding. Section 4.1.2 addresses the approach of grounding a complete model.

At the moment of writing this document, we are also evaluating the application of additional means for the maintenance of the vocabulary resulting from the grounding process. Figure 4.1 illustrates this in the form of a Semantic Wiki component, with the goal of providing the community of learners, teachers, experts, and users in general, with the means to collaboratively evolve the vocabulary. For the purpose of evaluating the convenience of such approach, we have designed a user study to be conducted with the WP6 partners. Table 4.1 summarizes the candidate systems to be evaluated in the user study, from which a subset will be eventually selected.

Table 4.1: State of the Art Semantic Wikis and Collaborative Ontology Development systems.

	Semantic Wikis				CODs		
	SMW	SMW+	OntoWiki	KiWi	Knoodl	WebProtégé	Col. Protége
Edit	Text	Text, Visual	Text, Visual	Visual	Visual	Visual	Visual
View	Text	Text, Visual	Text, Visual	Visual	Visual	Visual	Visual
Versioning / History	Yes	Yes	?	Yes	Yes	?	?
Access Control	?	Yes	Yes	Yes	Yes	Yes	?
Taxonomy	Yes	Yes	Yes	?	Yes	Yes	Yes
Instances	Yes	Yes	Yes	?	Yes	Yes	Yes
Relations	Yes	Yes	Yes	?	Yes	Yes	Yes
Properties	Yes	Yes	Yes	?	Yes	Yes	Yes
Constrains	No	No ?	?	?	Yes	Yes	Yes
Import of ontologies	Yes	Yes	?	?	Yes	Yes	?
Language	OWL/RDF	OWL/RDF	RDF	?	OWL/RDF, XSD	OWL	OWL
SPARQL	Yes	Yes	Yes	?	Yes	Yes (?)	Yes (?)
Underlying repository (Sesame, Jena)	?	?	?	Jena	Jena (?)	Protégé	Protégé
Licencing	GNU	GNU	?	Open Source	Free, licensing controlled by Revelytix	Open Source	Mozilla Public Licence
Special features				KiWi treats functionality and user interface components like any other content item that can collaboratively be created, edited, and used.			Concurrent mode: All changes made by one client are immediately visible by other clients.

4.1.1.1. Term grounding

The term grounding process is illustrated in Figure 4.1. At any time during model design, when an ingredient is being added to the model at hand, a request for grounding the term used to name such ingredient is issued from CM to ST. At this stage, ST first checks whether the term has been already grounded in the past. ST queries the thesauri associated to the domain vocabulary ontology contained in the Semantic Repository (SR) for any synonym of the current term.

If the term has been grounded before, the URI of the associated grounding contained in the domain vocabulary ontology is returned. Otherwise, the term needs to be normalized and a new grounding needs to be created and stored in the domain vocabulary ontology. For that purpose, ST first spell-checks the term. Then, a cascade search is started, which queries external resources WordNet, OpenCyc, and DBPedia for occurrences of the term. The different meanings of the term, including name and definition (if provided by the originating resource) are returned to the user via CM for (s)he to select the most appropriate one. If no grounding is found for a given term, ST notifies CM, which will ask the user to reformulate it. The meaning selected by the user is sent from CM to ST, which stores it in the domain vocabulary ontology using the URI of the resource providing such meaning in case of ontological resources (OpenCyc and DBPedia) or a new URI, local to the namespace of the domain vocabulary ontology in case it comes from linguistic resources (WordNet). Finally, the original term entered by the user is added to the thesauri in SR, associated to the grounded term.

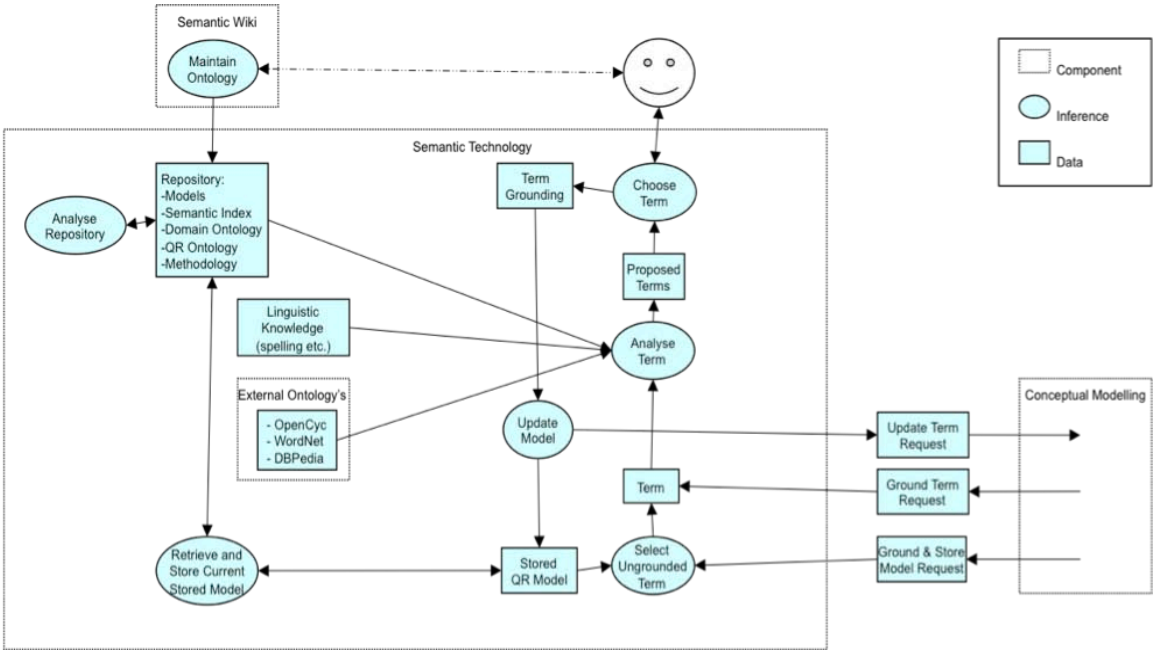

Figure 4.1: Term Grounding.

4.1.1.2. Model grounding

Model grounding occurs either when an explicit request is issued from CM to ST or when a model is saved (stored or updated) in SR, again upon CM invocation. In both cases, model grounding requires that the OWL representation of the QR model is stored in SR.

While the term grounding functionality is in charge of normalizing a term with respect to well-defined, established vocabularies acting as golden standards, model grounding allocates all the terms (which have been already grounded in the previously described term-grounding process) in the appropriate context of the domain vocabulary ontology, through their relations with the remaining terms in the model. For each model term, the following types of relations are considered

- **Consumption (*isA*) relations,** provided by the entity hierarchy of the QR model.
- **Role relations,** obtained by querying the model for ingredients like e.g. dependencies between entities and quantities.

Figure 4.2 shows the entity hierarchy (left-up) of an example model (communicating vessels) and a model fragment (left-down) describing the interactions between the ingredients of the model. On the right-hand side, the figure provides a representation of both *isA* and role relations extracted from the model. Ellipses represent ontology classes corresponding to the entities of the entity hierarchy while rectangles are ontology classes associated to quantity ingredients in the model. Role relations are extracted either from dependencies between entities e.g. *contains, from,* and *to* and part-whole relations between entities and their quantities e.g. *has*.

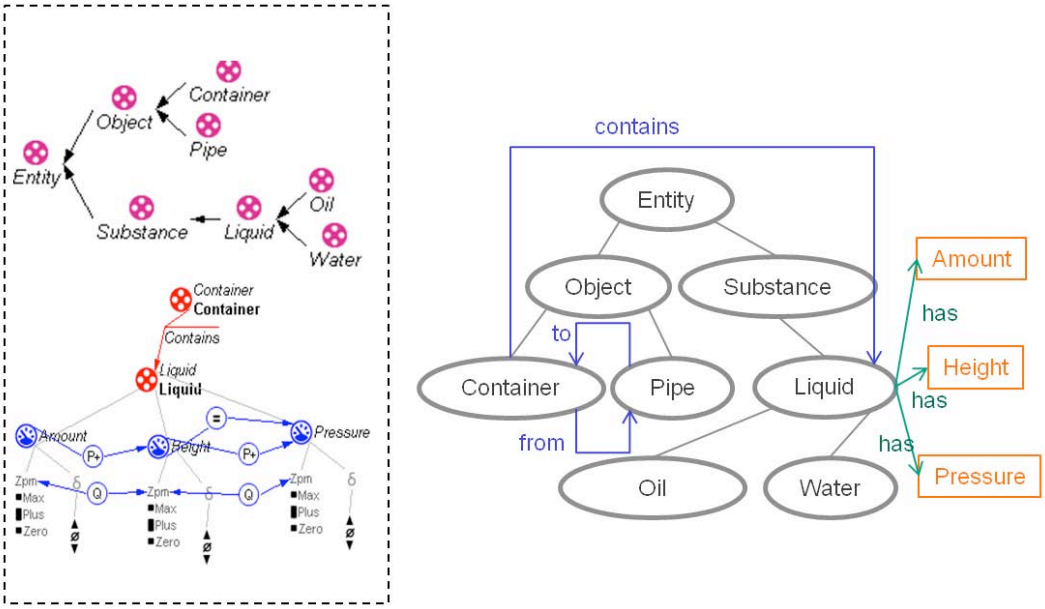

Figure 4.2: Model Grounding.

In a way analogous to term grounding, relations also need to be grounded in order to ground the overall model. The semantic consistency of *isA* relations is checked against external resources using taxonomic reasoning. In case inconsistencies are detected, ST notifies CM.

In the case of role relations, for each role relation identified, if it is already grounded, then nothing is done. Otherwise, it is lexically normalized and semantically validated with respect to the external resources acting as golden standards. ST sends the different possibilities provided by such resources, including name and description if available, to CM for the user to choose the most appropriate one. Finally, the original, user-generated name of the relation is added to the thesauri in SR, associated to the grounded relation. Like in the case of term grounding, the user can also discard all possibilities. In such event, the grounding is not created and therefore not stored in SR.

4.1.1.3. Multilingualism

Though the advantages of counting with a multilingual system will be appreciated also in the remaining use-cases described in section 4.1, the problem of producing multilingual resources needs to be addressed from the grounding perspective and is therefore described herein. At the moment of writing this document, two types of thesauri are considered to be kept in SR: synonym thesauri and language thesauri. The former contains, for each grounded term (identified by its URI), the different names used by users to refer to it. On the other hand, language thesauri contain the translation of each grounded term in the target languages of DynaLearn (fundamentally English and Portuguese). There will be one language thesauri per language and grounding management will also be in charge of multilingualism in the grounded terms.

4.1.2. Ontology-based feedback on model quality

Figure 4.3 illustrates the use-case associated to ontology-based feedback on model quality. Model quality is established fundamentally through similarities with golden standards. In this case, golden standards comprise, in increasing order of specificity:

- **External ontological and linguistic resources**. As in previous use-cases, our reference set in this aspect consists of OpenCYC and DBPedia and WordNet, respectively. These resources provide us with background domain knowledge, acting as a basis against which to validate more domain-specific, fine-grained knowledge representations like those stemming from particular QR models.

- **Models rated with high scores in the DynaLearn community** are trusted to be of high quality, serving as a reference for subsequent models.

- **Models authored by experts.** This kind of models reflects the expertise and knowledge of individuals with large amounts of knowledge in the area. This is the most significant population of models in order to establish the quality of any other model. Again, trust is a key concept in this regard, since we assume that models produced by experts are correct and provide design guidelines that should be followed by learners.

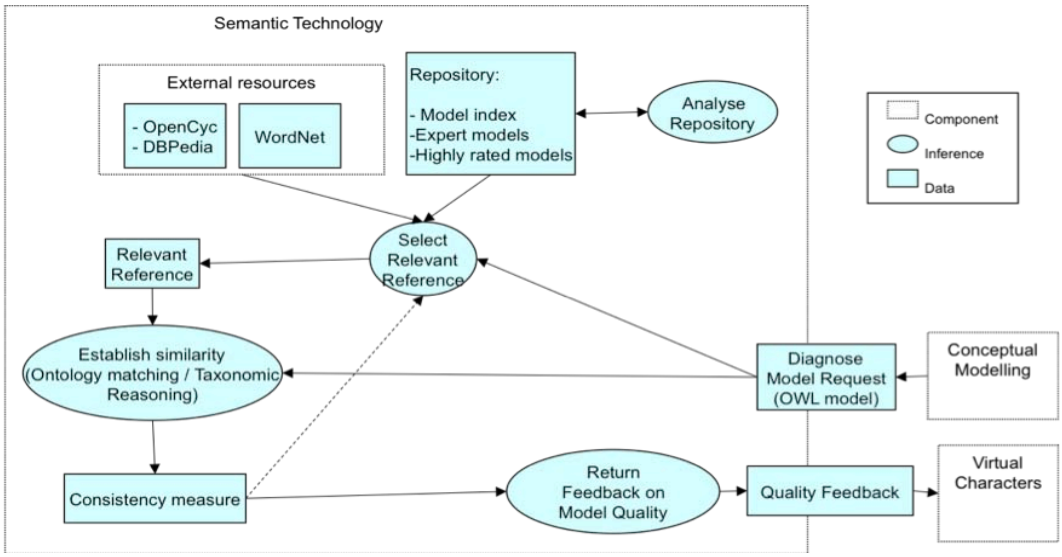

Figure 4.3: Ontology-based feedback on model quality.

This use-case is usually associated to the 'critic' VC, which highlights the deficiencies of the models authored by students during the learning process. When ST receives from CM a quality feedback request on a given model, a three-dimensional measure of the quality of the model is provided, based on its consistency with the golden standards. The values of each component of this measure are determined by the compliance of the model with, respectively: (i) the relevant external resources, (ii) the relevant highly rated models stored in SR, and (iii) the relevant expert models also stored in SR.

The relevance of the external resources for the model to be assessed is determined by extracting from SR the grounded terms of the model and then querying an ontology search engine like Watson[14] with those terms. Watson will then return a ranked list of the available ontological and linguistic resources, which match the terms, defining their relevance with respect to the model. Such search can be open, against all the ontologies available online, but also, and probably more focused and thus useful, constrained to a well-known, reference sample like the one proposed herein.

The QR model index ontology of SR classifies all the models stored in SR in terms of criteria like e.g. the topic from the curricula to which they belong. Additionally, two attributes are associated to each model in such ontology: *popularity* and *author expertise*. Thus, relevant

[14] http://watson.kmi.open.ac.uk/Overview.html

models rated high by the DynaLearn community are e.g. those, which belong to the same topic as the model to be assessed and have high-ranking figures. Likewise, the expert models, which are relevant for the model to be assessed, are those belonging to the same topic whose author has a high expertise. It will be possible to query the QR model index ontology and establish a threshold for both *popularity* and *author expertise* in order to filter the results.

Once the relevant resources are selected for each dimension of the quality measure, the model is compared against them using a combination of ontology matching and taxonomic reasoning techniques. Finally the three dimensions are aggregated in the form of feedback on the quality of the model and sent to the VC component in order to set up the appropriate dialogue with the user through the critic character.

4.1.3. Recommendations

We plan the development of recommendation techniques for the second half of project DynaLearn (tasks T4.3 and T4.4). However, in this document we introduce the main concepts underlying the design of the component, which will be refined at later stages. We observe one main type of recommendation driven by subjective community aspects and supported by **collaborative filtering** (CF) techniques. Through this type of recommendation, we aim at recommending model authors related models created by other members of the DynaLearn community, which could be relevant to them according to criteria like the following:

- *Users who liked model X also liked model Y*
- *Users who modelled X also modelled Y*
- *Users interested in topic X are also interested in topic Y*

A way to complement this kind of recommendations is by supporting also **recommendations based on the properties of the models under an objective perspective**. Models can be related with each other in terms of a number of dimensions, as described in [4]. By observing such dimensions, it is possible to relate models with each other in terms of generalization and specialization, analogy, inversion, order, and structural change. We anticipate two main use-cases of relating models with each other for recommendation purposes:

- **List all** models related to the current one in terms of each dimension (Figure 4.4).
- **Focus** on the most relevant models with respect to current one (Figure 4.5).

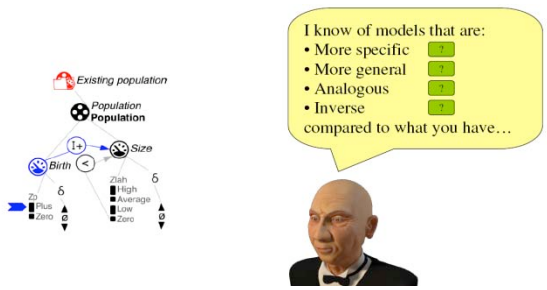

Figure 4.4: Objective recommendation use-case: List all.

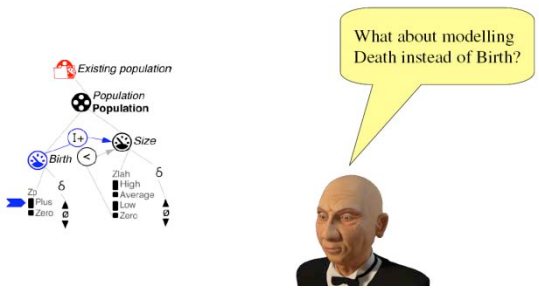

Figure 4.5: Objective recommendation use-case: Focus.

4.2. Architecture semantic technology component

Figure 4.6 shows the architecture proposed for the ST component addressing the three main use-cases grounding (Section 4.1), quality feedback (Section 4.2), and recommendation (Section 4.3). This architecture comprises a number of sub-components in order to address the requirements raised, and exposes an API for external components to access to their functionalities.

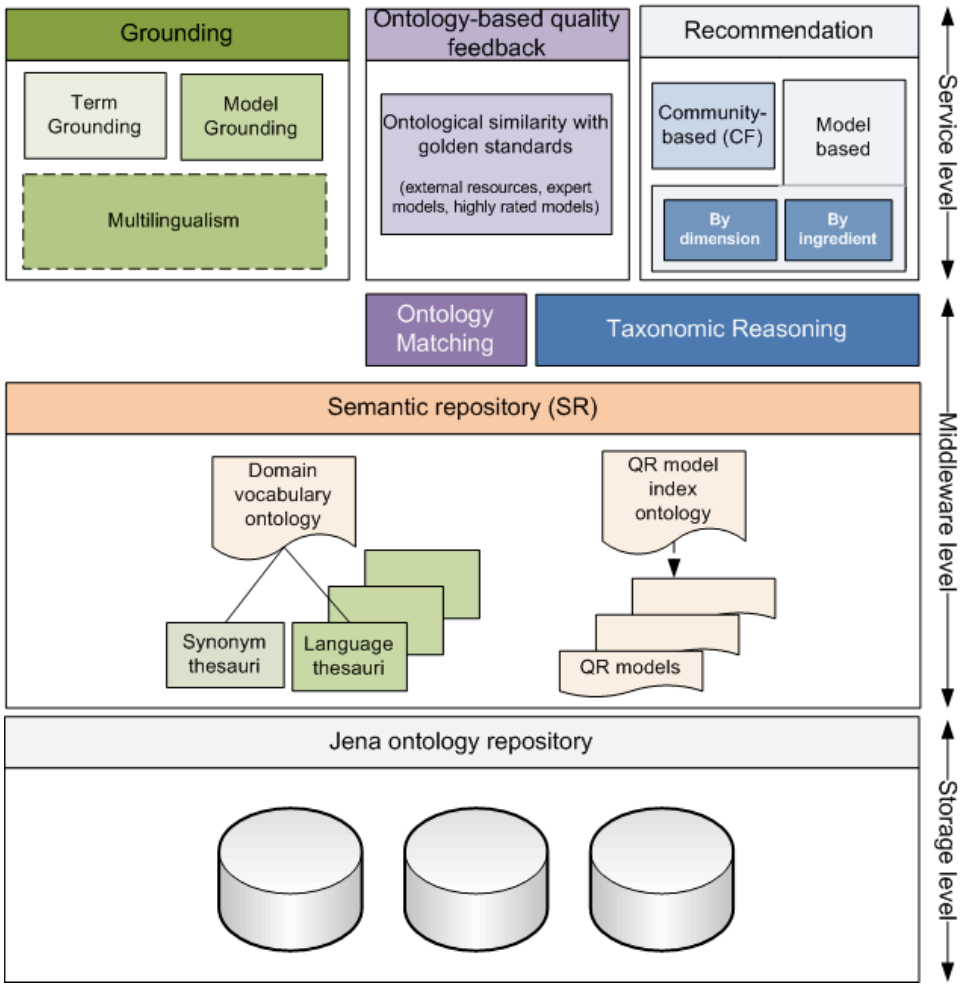

Figure 4.6: Architecture of the ST component.

The architecture follows a classical layered structure, where higher levels build on the functionalities provided by lower levels. The architecture components are structured in three main levels: storage, middleware, and service. Though the functionalities of all levels are equally exposed, their intended use is the following. The middleware level constitutes an overlay on top of the storage level, providing a higher level of abstraction for access to the repository and commodities like ontology matching and taxonomic reasoning, which will be horizontally used by several higher level components. On the other hand, the service level provides the remaining DynaLearn components (CM and VC) with access to the functionalities required in the different use-cases. Next, we describe each of these building blocks, their functionalities, and the APIs exposed.

4.2.1. Semantic repository (SR)

As introduced earlier, SR extends the underlying ontology repository, providing a higher level of abstraction on top of the functionalities of the ontology repository, built on Jena[15] for its OWL management capabilities. The goal of SR is to store and provide access to the ontological resources required by DynaLearn. Such resources can be classified as follows:

- **Domain vocabulary ontology**: It contains a semantic representation of the grounded terms and the relations between them, which have been extracted from the QR models upon their grounding and storage in SR. The ontology is complemented with two types of thesauri. First, a synonym thesaurus, including the different terms used by the members of the DynaLearn community in order to refer to the same (grounded) term. Second, a number of language thesauri, one per DynaLearn target language, with translations of each grounded term.

- **QR model index ontology**: This ontology classifies the QR models stored in SR according to some criteria. For the time being, the chosen classification criterion consists of the topic to which a given model belongs from amidst those in the curricula. The classification algorithm is illustrated in Figure 4.7. Each QR model is identified through its URI, thus treating them as first-class citizens. This ontology also stores some additional information, useful to appropriately handle the models, like *popularity* and *author expertise*, through service-level functionalities as in the recommendation component.

- **QR models** are represented as OWL ontologies, issues by CM at the moment of their creation. Each model is translated into an OWL ontology using the CM vocabulary, which stems from the Garp3 workbench.

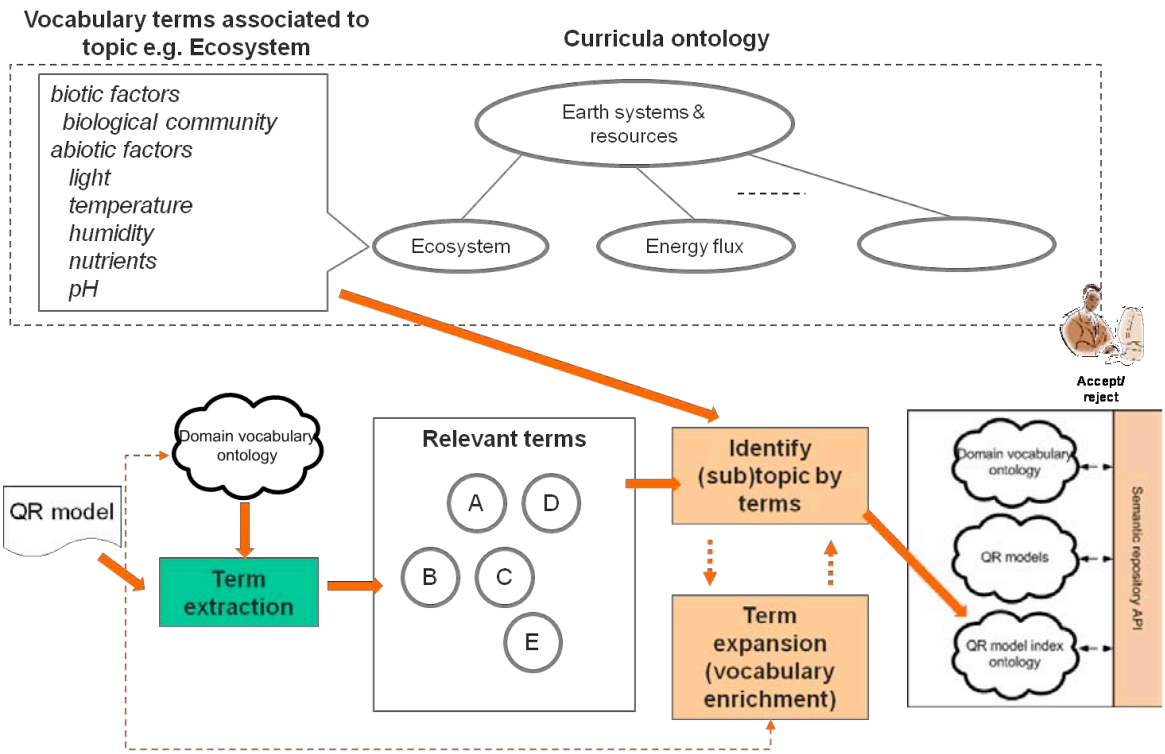

Figure 4.7: QR model classification (per curricula topic) algorithm.

[15] http://jena.sourceforge.net

4.2.1.1. Functionalities

The main functionalities of SR can be enumerated as follows:

- Store QR model
- Retrieve QR model
- Classify QR model
- Update QR model
- Delete QR model
- Retrieve all the role relations (object properties) of a class identified by its URI
- Retrieve all the isA relations of a class identified by its URI
- Retrieve all the QR models belonging to the given topics
- Retrieve all the QR models with popularity ranking greater/lower/equal than a given threshold (in a given topic)

4.2.1.2. API

- Store QR model:
  - Method name: **storeModel**
  - Input parameters: filename – the name of the ontology file
  - Output parameters: QRModel
  - Description: Store an ontology file of the QR model in the semantic repository and return one QRModel (an instance of class QRModel).
- Retrieve QR model:
  - Method name: **retrieveModel**
  - Input parameters: modelID – the model ID of the QR model in the semantic repository
  - Output parameters: QRModel
  - Description: Retrieve an ontology file of the QR model in the semantic repository and return one QRModel (an instance of class QRModel).
- Update QR model
  - Method name: **update**
  - Input parameters: filename – the name of the given file which includes the information for updating.
  - Output parameters: the status of the updating process. The value of status is {0 – successful, -1 – unsuccessful}

  - Description: Update the QR model with the information of the given file and return the status of the updating process.

- Delete QR model
  - Method name: **deleteQRModel**
  - Input parameters: modelID – the model ID of the QR model in the semantic repository
  - Output parameters: the status of the updating process. The value of status is {0 – successful, -1 – unsuccessful}
  - Description: Delete the QR model and return the status of the deleting process.

- Classify QR model
  - Method name: **classifyByTopic**
  - Description: This is a method in the class QRModel. This method classifies the QR model by topics: finds the topics which cover all of the relevant information (entity, quantity, model fragment...) in QR model, then adds QR model ID into the index.

- Retrieve all the role relations (object properties) of a class identified by its URI
  - Method name: **getAllObjectProperties**
  - Input parameters: uri – the URI of the given class
  - Output parameters: URI of all the object properties
  - Description: Retrieve all the role relations of a class identified by its URI.

- Retrieve all relevant information (entity, quantity, model fragment ...) of a model
  - Method name: **getRelevantInformation**
  - Output parameters: list of relevant information in this QR model
  - Description: Retrieve all relevant information (entity, quantity, model fragment ...) of a model.

- Retrieve all the isA relations of a class identified by its URI
  - Method name: **getAllSuperClasses**
  - Input parameters: uri – the URI of the given class
  - Output parameters: All the URI of the super classes
  - Description: Retrieve all the super classes of a class identified by its URI.

- Retrieve all the QR models belonging to the given topics
  - Method name: **listQRModelsByTopics**
  - Input parameters: listOfTopics – the list of the topics

- Output parameters: list of QR model IDs which have one or some topics in the list of the given topics
- Description: Return the model ID of all the QR models in the semantic repository which have some topics in the list of the given topics.

- Retrieve all QR models with popularity ranking greater/lower/equal than a given threshold (in a given topic)
  - Method name: **listQRModelsByPopularityRanking**
  - Input parameters:
    - Operator: 0 – equal, 1 – lower, 2 – greater
    - Value – the given threshold
    - Topic – the given topic (not mandatory)
  - Output parameters: The list of the model ID
  - Description: Return the model ID of all the QR models with popularity ranking greater/lower/equal than a given threshold (in a given topic).

4.2.1.3. Semantic Repository Sequence Diagram

This section illustrates the functionalities of the Semantic Repository. The sequence diagrams included herein show how the API described in the previous section implements the functionalities required for such use-cases.

A model is stored when CM sends the model to ST and request to ground or store model (Figure 4.8). Before storing in the semantic repository, the model is grounded. After all, ST gets all the relevant information e.g. entity, quantity, model fragment ... to classify the model by topic and update the index file with the information of the model.

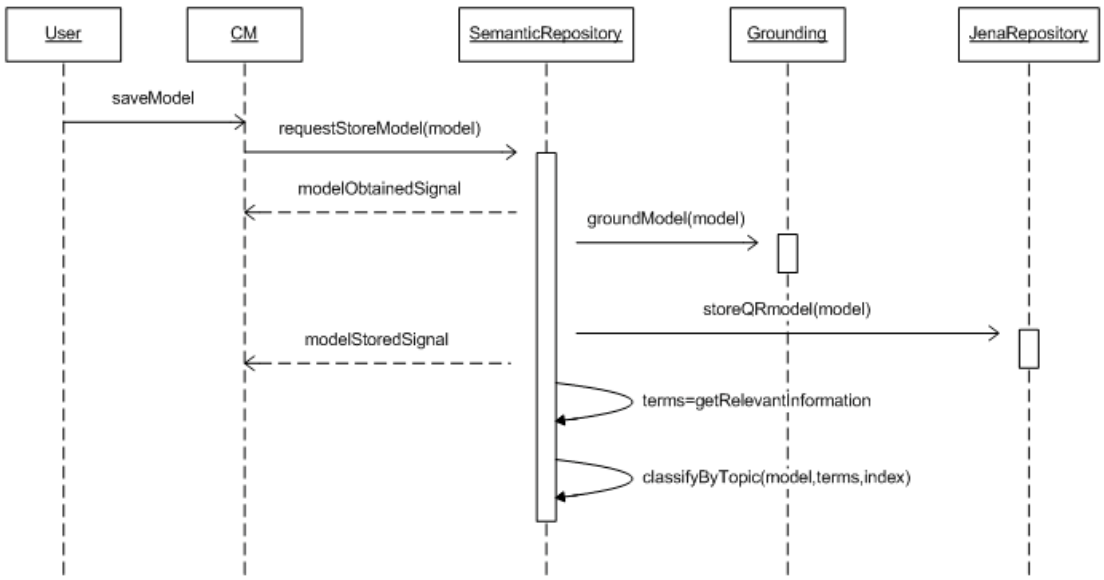

Figure 4.8: Store model sequence diagram.

To retrieve a model, CM needs to send model retrieve request with the model name (Figure 4.9). ST uses the input model name to retrieve the model in the semantic repository and send the model to CM.

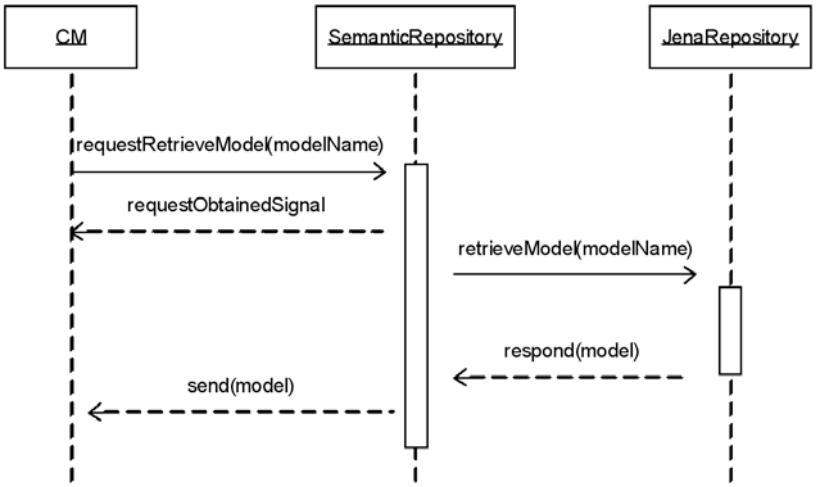

Figure 4.9: Retrieve model sequence diagram.

4.2.2. Grounding component

The grounding component is in charge of grounding the terms contained in each QR model to be kept in the domain vocabulary ontology in the semantic repository and also retrieving these terms in order to propose term grounding for the new terms in the model to the user. Moreover, it also provides a programmatic interface to do the grounding of the model, i.e. puts the grounded terms in the proper context of the domain vocabulary ontology after all the terms are properly grounded. Overall introduction of the concept as well as detailed description can be found in the Section 4.1.1. The grounding component also encapsulates the functionalities related to the support of multilingualism in the models, i.e. provides the access (store/retrieve) to the synonym thesauri and language thesauri as introduced in Section 4.1.1.3.

4.2.2.1. Functionalities

The main functionalities of the Grounding component include the following:

- **Ground term**
- **Ground model**
- **Store term grounding in domain vocabulary**
- **Retrieve term grounding from domain vocabulary**
- **Store synonym in synonym thesaurus**
- **Retrieve synonym from synonym thesaurus**
- **Store term translation in language thesauri**
- **Retrieve term translation from language thesauri**
- **...**

4.2.2.2. API

- Term
  - Class name: **Term**
  - Member variables:
    - Term – String containing term itself
    - Language – identifier of the language of this term
  - Description: Encapsulates the term and identifier of its language.
- Is term normalized?
  - Method name: **isTermNormalized**
  - Input parameters: term to be analyzed
  - Output parameters: true or false depending on whether term is normalized
  - Description: Indicate whether term is normalized.
- Proposed normalizations for a term
  - Method name: **getTermNormalizations**
  - Input parameters: term to be analyzed
  - Output parameters: set of proposed normalizations of the term
  - Description: Provide a set of proposed normalized terms for the proposed term.
- Is term grounded?
  - Method name: **isTermGrounded**
  - Input parameters: term to be analyzed
  - Output parameters: URI of the grounding or NULL is no grounding found
  - Description: Indicate whether the term has been found in the SR, i.e. is grounded.
- Proposed groundings for a term
  - Method name: **getTermGroundings**
  - Input parameters: term to be analyzed
  - Output parameters: set of proposed groundings of the term
  - Description: Provide a set of proposed grounded terms for the proposed term.
- Ground term
  - Method name: **groundTerm**

  - Input parameters: term T to be grounded, reference to the source to which T will be grounded
  - Output parameters: URI of the grounding of T or exception in case of an error
  - Description: Ground a term to the associated grounding.

- Is model grounded?
  - Method name: **isModelGrounded**
  - Input parameters: model to be analyzed
  - Output parameters: URI of the grounding or NULL is no grounding found
  - Description: Indicate whether the model has been found in the SR, i.e. is grounded.

- Proposed groundings for a model
  - Method name: **getModelGroundings**
  - Input parameters: model to be analyzed
  - Output parameters: set of proposed groundings of the model
  - Description: Provide a set of the proposed grounded models for the proposed model.

- Ground model
  - Method name: **groundModel**
  - Input parameters: model M to be grounded, reference to the source to which M will be grounded
  - Output parameters: URI of the grounding of M or exception in case of an error
  - Description: Ground a model to the associated grounding.

- Store term grounding in domain vocabulary
  - Method name: **storeGrounding**
  - Input parameters: normalized term T, a grounding of T
  - Output parameters: true of T or exception in case of an error
  - Description: Ground a term to the associated grounding and store it in the domain vocabulary.

- Retrieve term grounding in domain vocabulary
  - Method name: **getGrounding**
  - Input parameters: normalized term T
  - Output parameters: URI of the associated grounding or an exception
  - Description: Retrieve the associated grounding to the normalized term.

- Store synonym in the synonym thesaurus
  - Method name: **storeSynonym**
  - Input parameters: groundedTerm1, groundedTerm2 - pair of synonyms
  - Output parameters: true or an exception
  - Description: Stores the pair of synonym grounded terms in the synonym thesaurus.
- Retrieve synonym from the synonym thesaurus
  - Method name: **getSynonymSet**
  - Input parameters: groundedTerm – an analyzed grounded term
  - Output parameters: set of synonyms
  - Description: Retrieves the set of terms that are synonyms to the input term (including the input term).
- Store term translation in the language thesauri
  - Method name: **storeTranslation**
  - Input parameters: term1, term2 – a pair of grounded terms to be stored
  - Output parameters: true or an exception in case of an error
  - Description: Stores translation pair in the language thesauri.
- Retrieve term translation from the language thesauri
  - Method name: **getTranslation**
  - Input parameters: term – a grounded term that's translation we want to obtain, language – language to which the term should be translated
  - Output parameters: grounded term in the target language
  - Description: Retrieve the translation of the term from the language thesauri.

4.2.2.3. Grounding Component Sequence Diagrams

This section describes the functionalities of the Model Grounding ST of the Technology that we have first introduced in the use-cases described in Section 4.1.1. The sequence diagram (Figure 4.10) included here shows how the API implements the functionalities required for these use-cases.

Each grounding is expected to be done in two stages: first, for each user-chosen term, we have to find a proper normalization in the language of the model. This has to be done with the help of the online and/or spell-checking tools and a result will be a term consisting of valid forms of the words from the vocabulary. If user provides a correct version of the word already in the beginning, we accept it and skip the rest of this part.

In the second stage, we will ask user to indicate proper grounding of the term. This is be done by offering a list of possibilities found by Grounding Component. User will choose one of them or create a new grounding and this will be uploaded to the Semantic Repository.

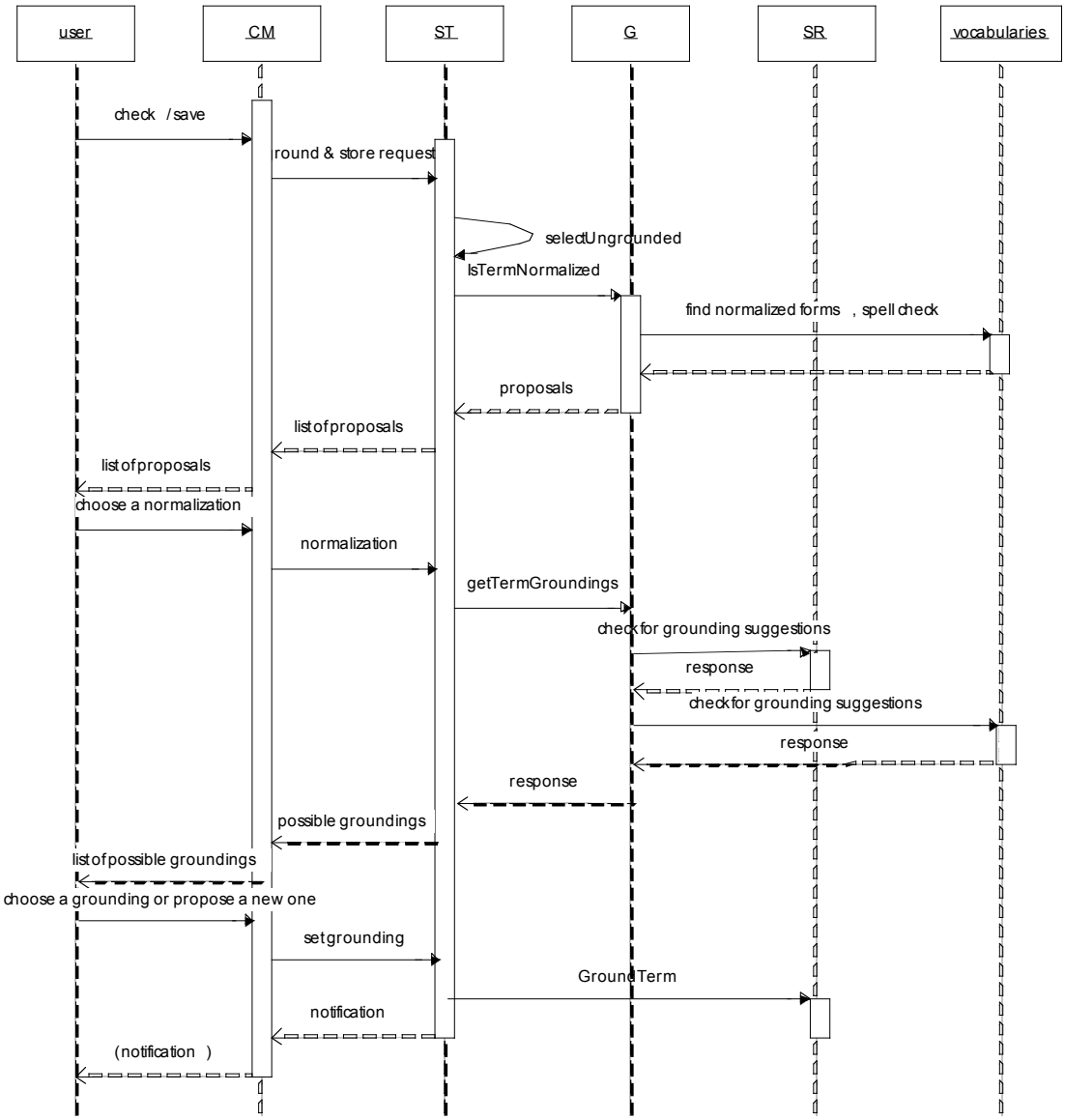

Figure 4.10: Retrieve model sequence diagram.

4.2.3. Ontology-based quality feedback component

The ontology-based quality feedback component is providing functionalities related to the concept of quality feedback already described in the Section 4.1.2. The component provides both qualitative and quantitative feedback, the former one in the form a single number providing a computed rank of the model, the latter in the form of a list of the relevant models and external resources found by component.

4.2.3.1. Functionalities

The main functionalities of the ontology-based quality feedback component include the following:

- **Provide quality feedback on model**
- **Check model quality with respect to external resources**
- **Check model quality with respect to highly rated models**
- **Check model quality with respect to expert model**
- **Compose 3-D quality measure for a model**
- **Select relevant external resources for a model**
- **Select relevant highly rated models for a model**
- **Select relevant expert models for a model**
- **Obtain overall matching value of two ontologies**
- **Obtain matching value of two entities from two different ontologies**
- **...**

4.2.3.2. API

- Rank the quality of the model with respect to external resources
  - Method name: **rankByExternalResources**
  - Input parameters: model – model to be ranked
  - Output parameters: rank in the interval [0,1], where 1 is the best
  - Description: Ranks the model according to the knowledge taken from the external resources (OpenCYC, DBPedia and WordNet). Returns one aggregated value.
- Rank the quality of the model with respect to other model
  - Method name: **rankByAnotherModel**
  - Input parameters: method – method used: 0 – ontology matching, 1 – pattern similarities, 2 – aggregated value made from former ones; model – model to be ranked; goldenStandard – golden standard model
  - Output parameters: rank in the interval [0,1], where 1 is the best
  - Description: Ranks the model according to the knowledge taken from the external resources (OpenCYC, DBPedia and WordNet). Returns one aggregated value.
- Rank the quality of the model with respect to all three criteria (external resources, highly rated models, expert model)

  - Method name: **getComposedRank**
  - Input parameters: method – method used: 0 – ontology matching, 1 – pattern similarities, 2 – aggregated value made from former ones; model – model to be ranked; goldenStandard – golden standard model
  - Output parameters: rank in the interval [0,1], where 1 is the best
  - Description: Ranks the model according to the knowledge taken from the external resources (OpenCYC, DBPedia and WordNet). Returns one aggregated value.

- Select relevant external resources for a model
  - Method name: **getRelevantExternalResources**
  - Input parameters: model – model to be compared
  - Output parameters: list of relevant resources found
  - Description: Finds the most relevant external resources (in OpenCYC, DBPedia and WordNet) and returns list of them.

- Select relevant models for a model
  - Method name: **getRelevantModels**
  - Input parameters: model – model to be compared, filter – holding values 0 for popularity filtering, 1 – for filtering based on expert opinion, 2 – for mixed approach based on former
  - Output parameters: list of relevant models found
  - Description: Finds the most relevant models for the quality feedback and returns list of them.

- Rank the quality of the entity with respect to other entity
  - Method name: **getEntityMatchingValue**
  - Input parameters: entity1, entity2 – two compared entities, method – identifier specifying the method used for comparison (0 – ontology matching algorithm, 1 – pattern/(anti-)pattern approach)
  - Output parameters: rank in the interval [0,1], where 1 is the best match
  - Description: Measure the similarity between two entities in the models.

4.2.3.3. Ontology-Based Quality Feedback Component Sequence Diagrams

This section describes the functionalities of the Ontology-Based Quality Feedback component of the ST part that we have first introduced in the use-cases described in Sections 4.1.2. The sequence diagram included here shows how the API implements the functionalities required for these use-cases.

This module has two basic goals: the first one is to rank a model with the use of other models and/or external resources (Figure 4.11). The second goal is to create a set of recommendations (Figure 4.12).

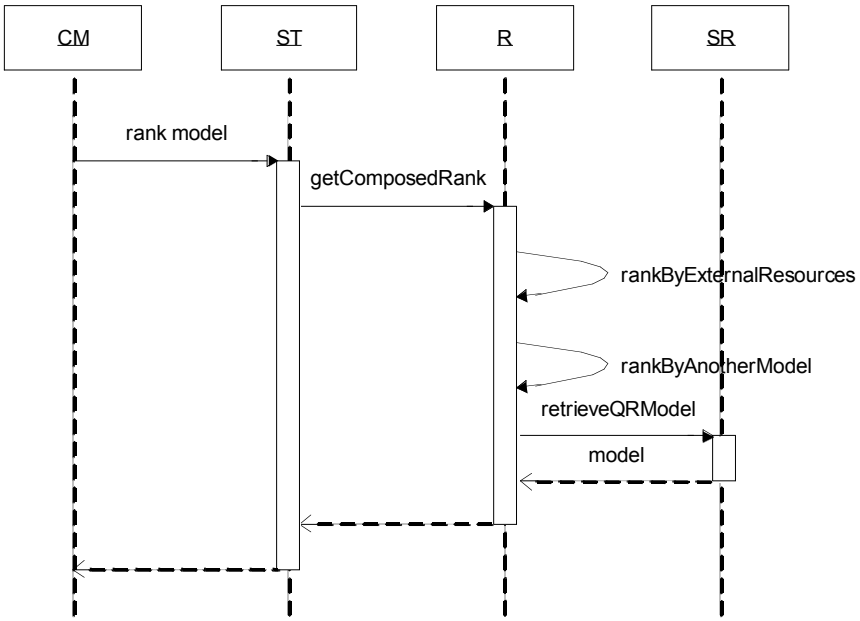

Figure 4.11: Sequence diagram showing process of counting a rank of a model.

The task of counting a rank is task for two main functions: rankByExternalResources, that will use external resources to collect relevant information and count the rank, the other is rankByAnotherModel, that will, depending on the given parameters, count the similarities and differences for another model created by an expert or other learners.

The later task, to create a list of recommendation, for a VC, is very similar in nature. The basic difference is that instead of counting and producing a single number, we generate a structure, that is describing the similarities and differences found in more detail.

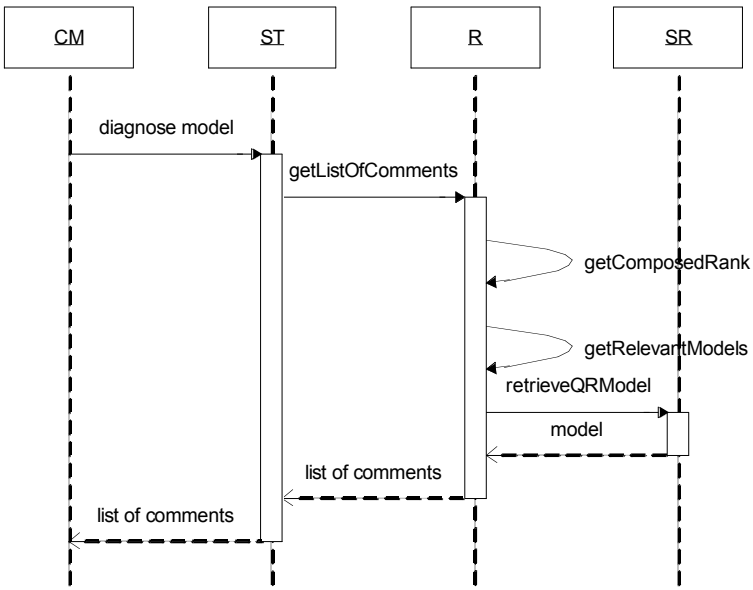

Figure 4.12: Sequence diagram showing process of creating a report about quality of the model.

4.2.4. Recommendation component

As introduced in the section 4.1.3, the goal of the recommendation component is to provide users with recommendations based on the properties of the models under an object perspective. The related models for recommendation are suggested in two ways:

- List all models related to the current one in terms of each dimension (e.g. generalization, specialization, analogy, inversion, order and structural change).
- Focus on the most relevant models with respect to the current one.

4.2.4.1. Functionalities

The main functionalities of the recommendation component include the following:

- **Retrieve socially akin models** to the present one e.g. *users who liked model X also liked model Y, users who authored a model of topic X also created a model of topic Y, users who modelled X, also modelled Y…*
- **Retrieve all the QR models of a given user** (user who modelled model X also modelled model Y)
- **Retrieve all the interesting topics of a given user** (user who is interested in topic X also is interested in topic Y)
- **Retrieve all the topics of the user's models**
- **Retrieve all models related with a model in terms of the dimensions** (see Figure 4.13) e.g. more specific, more generic, analogous or inverse models.
- **Propose model focus e.g. a ranked list of models which might be interesting for the user to model after (s)he finishes the current one** (see Figure 4.14)

4.2.4.2. API

- Retrieve the list of QR models which are similar to the present one
  - Method name: **listAkinQRModels**
  - Output parameters: list of QR model IDs which are similar to the present one
  - Description: list of QR model IDs which are similar to the present one.
- Retrieve the list of QR models made by a given user
  - Method name: **listQRModelsByCreator**
  - Input parameters: username – a given creator
  - Output parameters: list of QR model IDs made by a given user
  - Description: Return the model ID of all the QR models in the semantic repository made by a given user.
- Retrieve the list of topics which have greater or equal interest level with a given topic from a given user

    - Method name: **listInterestingTopics**
    - Input parameters:
        - Username – a given creator
        - Topic – a given topic
    - Output parameters: list of topics made by a given user
    - Description: Return the list of topics which have greater or equal interest level with a given topic from a given user.
- Retrieve the list of QR models which the user is interested in
    - Method name: **listTopicsRelatedWithCreator**
    - Input parameters: username – a given user
    - Output parameters: list of topics which is related with the given user's models
    - Description: Return the list of topics which is related with the given user's models.
- Retrieve all the QR models with a model in terms of the dimensions
    - Method name: **listAllQRModelsByDimensions**
    - Input parameters:
        - Type of dimensions – the value for the recommendation process
        - Relevant information of an user model
    - Output parameters: list of QR model IDs
    - Description: Return model ID of all the QR models with a model in terms of the dimensions
- Focus on the relevant QR models with respect to the current QR model
    - Method name: **focusOnRelevantQRModels**
    - Input parameters: relevant information of an user model
    - Output parameters: list of QR model IDs which are relevant to the current one
    - Description: Return all the relevant QR models with respect to the current one

4.2.4.3. Recommendation Sequence Diagrams

This section illustrates the functionalities of the recommendation use-case first introduced in the use-case described in Section 4.1.3. The sequence diagrams included herein show how the API described in the previous section implements the functionalities required for such use-cases.

CM has two ways to offer recommendation to a user:

- List all models related to the current one in the terms of each dimension (Figure 4.13): CM sends ST a model and a dimension, which is selected by a user. To provide the recommendations, ST selects the relevant information in the model and gets all models,

which have the similar information to the relevant information in terms of dimension. The list of the selected models and the dimension are sent to VC.

- Focus on the most relevant models with respect to current one (Figure 4.14): CM sends the model and the request recommendations to ST. ST gets all the relevant information of the model and use the information to retrieve all the most relevant models in the semantic repository. Those models are sent to VC to present to user.

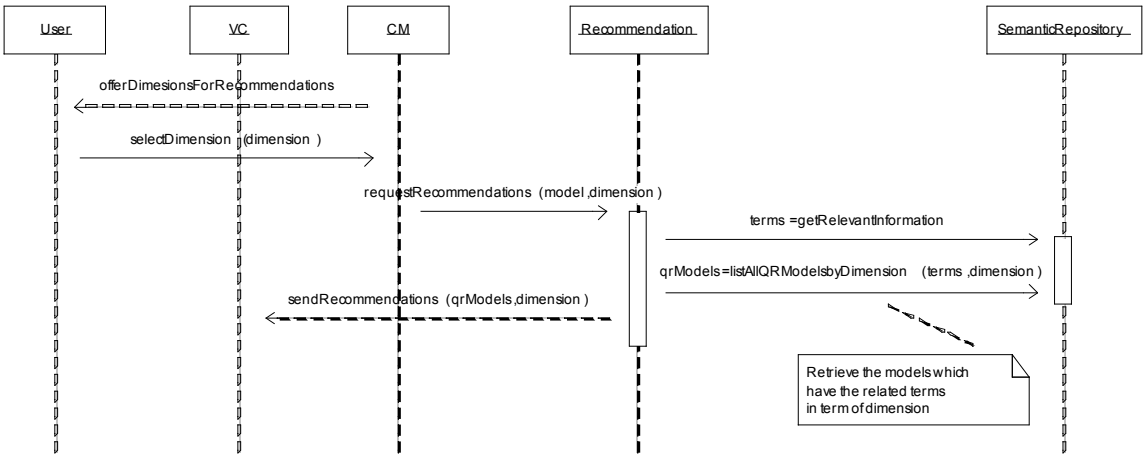

Figure 4.13: List all models in terms of dimension.

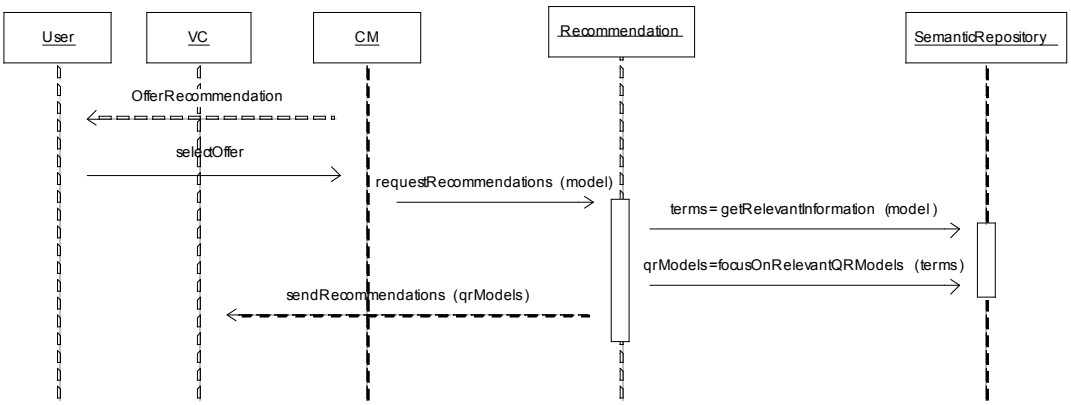

Figure 4.14: Focus on the most relevant models.

4.3. Planning semantic technology development

Figure 4.15 shows the development plan for the Semantic Technology component throughout DynaLearn. Next, Table 4.2 summarizes such contributions.

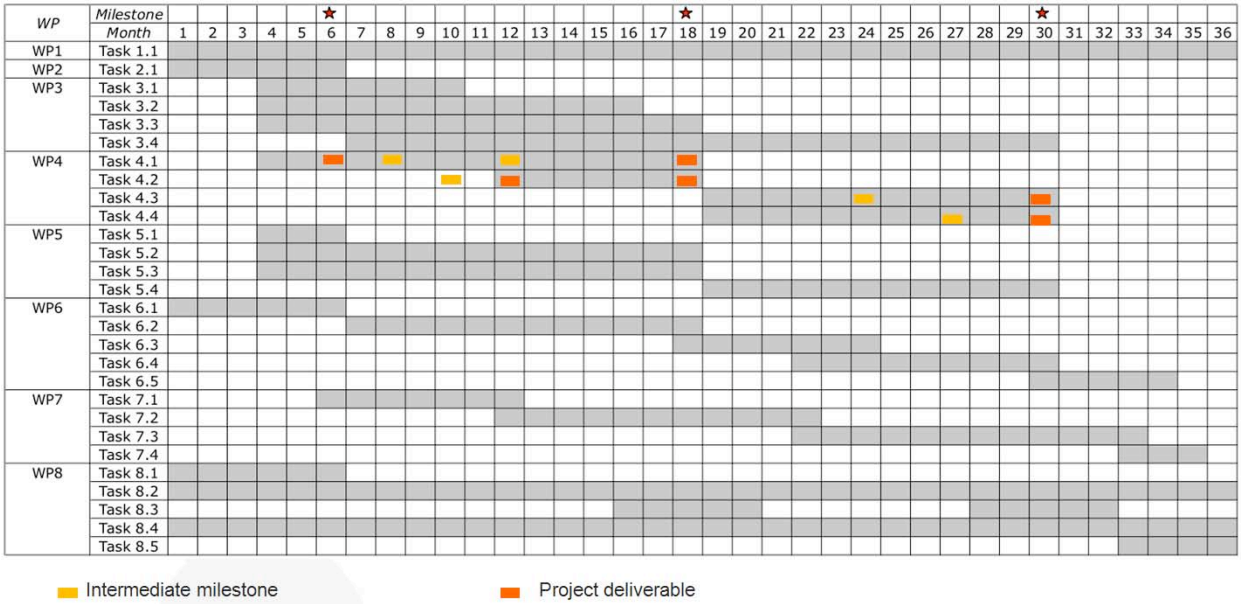

Figure 4.15: Milestones for the ST component development.

Table 4.2: Detailed ST development plan.

Month	Milestone	Milestone type	Milestone description
6	M4.1.1	Intermediate	**SR** basic functionalities accessible via sockets (store/retrieve model, classify model vs. topic)
8	M4.1.2	Intermediate	**Vocabulary creation**: Synthesis of ontology model (intermediate representation) and term grounding
10	M4.2.1	Intermediate	**Quality feedback:** Ontology matching intermediate representations vs. golden standards (experimental results plus benchmarks defined)
12	M4.1.3	Intermediate	**SR**: Advanced functionalities including model classification through models dimensions (preliminary) **Vocabulary Creation:** Model (through term) grounding in vocabulary ontology
	M4.2.2	Intermediate	**Quality feedback:** ontology-based feedback vs. golden standards (preliminary)
18	M4.1.5	Deliverable	**SR:** final software
	M4.2.3	Deliverable	**Quality feedback:** final software
24	M4.3.1	Intermediate	**CF** for community-based recommendation (preliminary)
27	M4.4.1	Intermediate	**Taxonomic reasoning** (preliminary)
30	M4.3.2	Deliverable	**CF** for community-based recommendation (final)
	M4.4.2	Deliverable	**Taxonomic reasoning** (final)

5. Virtual character component

During the technical planning of the Avatar concepts for the DynaLearn project the question came up of how to implement platform independent 3D characters that fit into the GARP3 editing environment for a consistent user experience. Besides Augsburg's own character animation technology based on the freely available Horde3D GameEngine and Microsoft Agent (which is quite far developed concerning virtual agent tasks but limited to the Windows platform), Adobe Flash came into focus especially with its youngest trends towards rich desktop applications beyond web browser windows. Furthermore, a draft was required for an intermodular communication technology to exchange data and commands between the GARP main application and the enriching components like the VC.

5.1. Design dialogue functionality for characters

5.1.1. Curriculum scripts

The aim is to rely on curriculum scripts (see [7]), which are a combination of pre-authored and dynamically created material. Curriculum scripts bear the advantage that they enable subject matter experts to contribute pedagogical material. Furthermore, they minimize the risk that there is no appropriate content for the characters to present because of difficulties of the ST and CM to provide that content.

Curriculum scripts provide the characters with content to be communicated to the learner, such as a problem to be solved by the student, an ideal solution, explanations, hints, frequent misconceptions etc. The content may be textual paragraphs that are annotated with actions for the characters or qualitative models, for example, one to be extended by the student and another one, which represents the expected (ideal) solution by the teacher. The instructional material is then used by the characters as a basis for a dialogue with a student.

Depending on a student's feedback, a different path is taken in the curriculum. We will allow for simple user input either via menus or via typed natural language input. A semantic parser will be used to define rules for interpreting simple typed user input (e.g. asking yes/no questions, asking if a quantity increases or decreases, and other multiple choice questions).

The following example presents a part of a curriculum script with a problem stated by the agents and two hints that may be given in case the student gets stuck.

> Problem: I've seen a lake that was polluted by the sewage produced in a village nearby. As a consequence, the water was not clean, children in the village got diarrhea and the fisherman complain the amount of fishes is decreasing. Let's create a model to support understanding about the effects of sewage produced in the village being thrown into the lake.
>
> Hint 1: First thing is to identify the main concepts in your story.
>
> Hint 2: Criteria to identify entities include physical objects or abstract things that have properties that could be quantified.

Please note that the agents are not necessarily involved in all parts of the curriculum. For example, the problem could be stated in a traditional manner while the agent just comes into play to provide hints. Furthermore, several agents may be involved in a script. For example, the problem could be presented by two different expert agents.

We intend to represent curriculum scripts using SceneMaker [8], a tool for authoring interactive performances with embodied conversational agents. In SceneMaker, a scene flow is modeled by means of a finite state machine in which states refer to scenes with characters. In DynaLearn, such a state could be the presentation of a problem by the agent (as shown above). The content of the scenes is either completely pre-scripted by a human author or generated automatically on-the-fly. Based on our earlier work [9], we will use a hierarchical planner in combination with a simple template-based generator to automatically create presentations.

5.1.2. Character roles

DynaLearn will support different character roles (such as expert, learning companion and others) and different dialogue styles (non-interactive dialogues of character teams and interactive dialogues between a student and one or more virtual characters).

In the previous paragraph, we described curriculum scripts to represent the overall structure of a course unit. Here we will concentrate on more elementary dialogue functions as part of a curriculum unit. Essentially, the characters may

- Provide basic help when
    - The student has a particular question on the use of the software (How to?)
    - The student has a particular question on a model (What is? Why?)
- Engage the student into a quiz
- Provide an evaluation of a model created by the student in terms
    - of a diagnosis/critic
    - of a comparison

In addition, students may teach their agents and observe their performance in a quiz (dialogue between multiple characters, such as a quiz master and two students) and engage in a chat with other (human) learners via virtual peer learners.

The content is presented by the characters based on input provided by CM and ST or based on pre-authored material contained in the curriculum scripts.

The following sections give an overview of how the characters are integrated into the global architecture whereby we assume that the content is presented based on input provided by CM and ST (even though pre-authored material from the curriculum scripts may be used as a fallback solution).

5.2. Virtual characters design – content (phase 1)

This section describes the characters due by project M18.

5.2.1. Basic help – What is?

The What is? help gives the learner information on a specific content element of the model. Figure 5.1 shows this interaction concept.

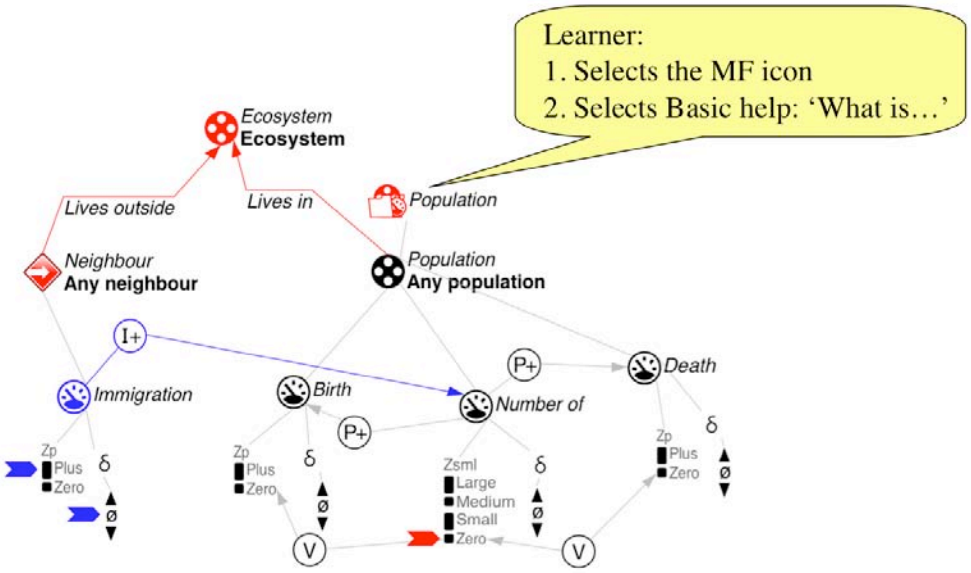

Figure 5.1: Interaction concept for model context sensitive help.

5.2.1.1. Components

The following components are used for this feature:

- The GARP workbench (WB) receives the help request through user interaction and highlighted model fragments.
- Actual help contents are built considering the highlighted model content.
- VC Behaviour Engine receives the explanation contents and constructs or updates a dialogue plan for the presentation. The avatar presentation is launched finally.

5.2.1.2. API

Data structures shared with the CM component contain:

- Topic Help Request (WB → CM)
    - Parameter: Topic identifier or model context for addressing the relevant contents
- Content Result / Help Display Command (CM → VC)
    - Parameter: Actual content to be displayed

5.2.2. Basic help – How to use?

The How? help gives the user hints on general usage of the software. The students request help on a specific functionality or modelling concepts.

5.2.2.1. Components

The following components shown in Figure 5.2 are used for this feature:

- VC's Interaction Manager fetches the help request from the user.
- Input analyzers transform the input into topics and determine the Help Request.
- The help request is sent to the ST where content help is produced based on models, semantics and ontologies.
- VC Behaviour Engine constructs or updates a dialogue plan for the presentation of the received solution or help information and launches an avatar presentation.
- Construction of dialogs and presentations may be influenced by previous interactions logged in the user model or pedagogical strategies and targets.

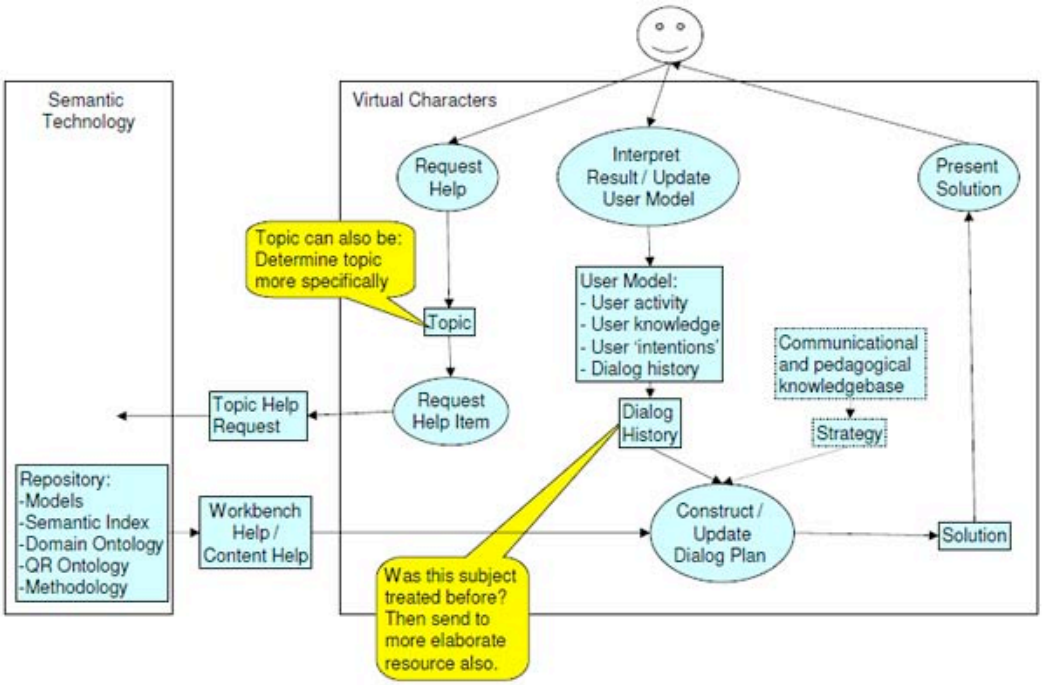

Figure 5.2: Architectural view on Basic Help functionality.

5.2.2.2. API

Data structures shared with the CM component contain:

- Topic Help Request (CM ← VC)
  - Parameter: Topic identifier for addressing the relevant contents
- Content Result (CM → VC)
  - Parameter: Actual content to be displayed

5.2.3. Basic help – Why?

The Why? help is the most detailed version of the basic help functionality. The basic idea is illustrated in Figure 5.3. It not only considers the model elements but also dependencies and effects resulting from the user's modelling. The system should explain reactions and dependencies to help the user to understand the constructions and its outcomes. This may lead users to find and correct mistakes themselves.

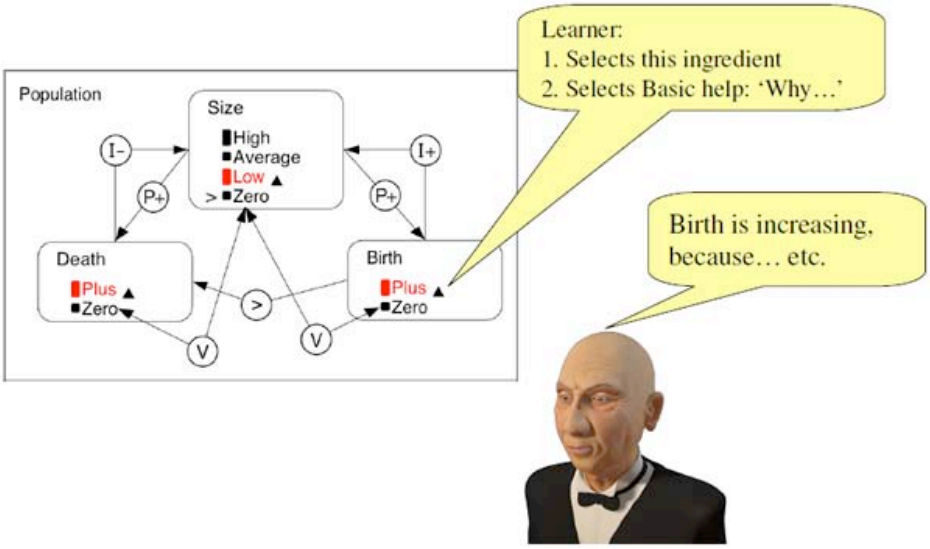

Figure 5.3: Concept of the Why? help functionality.

5.2.3.1. Components

The following components are used for this feature:

- The CM workbench receives the help request through user interaction and highlighted model fragments.
- The request is transformed into actual help contents using additional components and functions from CM or ST that are able to interpret the modelled dependencies.
- The avatar presentation is launched as soon as VC Behaviour Engine constructed a dialogue plan for the presentation from the received contents.

5.2.3.2. API

Data structures shared with the CM component contain:

- Topic Help Request (WB → CM)
    - Parameter: Topic identifier or model context for addressing the relevant contents
- Content Result / Help Display Command (CM → VC)
    - Parameter: Actual content to be displayed

5.2.4. Teachable agent / Learning by teaching

The possibility to train virtual characters follows the idea of learning by teaching. While training avatars the student has to look into the learning subjects, which improve his or her own knowledge in turn. Furthermore, the teachable agent is related to the learning companion who represents the teachable agent of another student.

It is probably a good idea to restrict the availability of the teachable agent to specific use-levels of the CM component. For instance, there is no use to have a teachable agent for the concept

mapping activity of the lowest use-level, because such a representation does not allow the underlying software to make sensible inferences about systems behaviour. Also at the most complex level the use of a teachable agent is disputable. The expressions made by learner at that level can become very complex, and evaluating the expertise acquired by the teachable agent hence becomes an endeavour that seems out of scope of the idea 'learning by teaching'. The notion of a teachable agent will be most valuable for the use-levels 2 (basic causal model), (basic causal model with state-graph), and 3 (causal differentiation) in which the expressions created by learners focus primarily on the causal model explaining the target system behaviour within a single state of behaviour (Section 3). In DynaLearn, we will therefore first develop the teachable agent for use-level 2, and after completion augment it to include use-level 3 and 4.

Teachable agents behave like personal pets and learn automatically when the user uses Garp3. Learning events do not happen explicitly but all the time during the modelling processes. To allow the implementation of achievement based customizations for the avatars to make inter-avatar competitions more entertaining, some internal notifications may be sent to the virtual characters at key interaction or editing events.

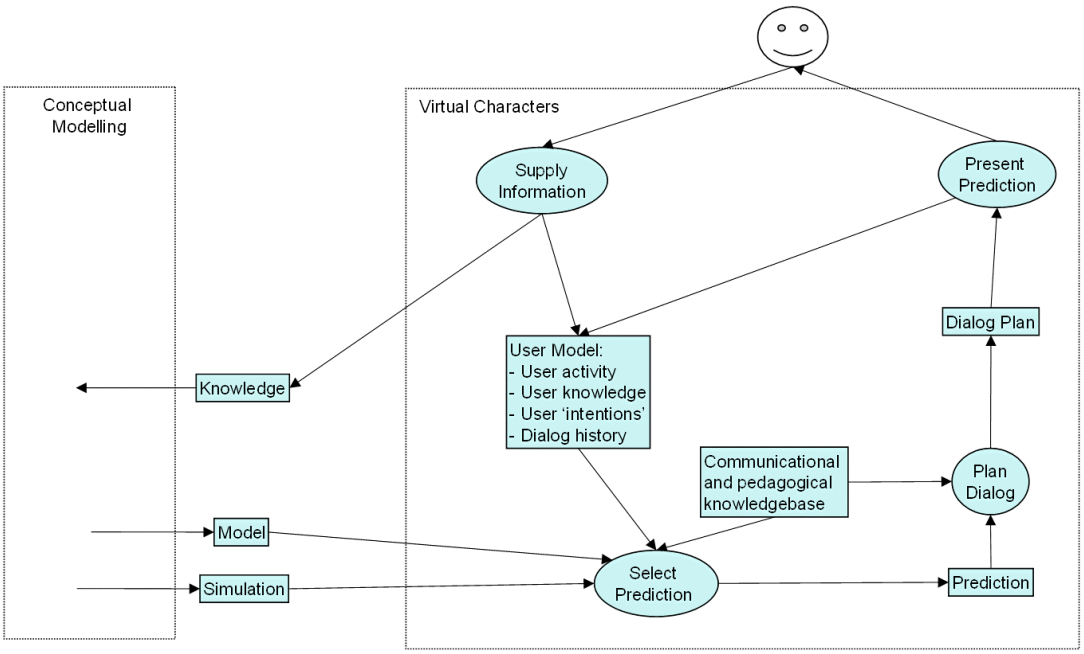

Figure 5.4: Technological model for teachable agents.

5.2.4.1. Components

The following components are used for this feature as shown in Figure 5.4:

- The CM workbench notifies the VC when key events occur. An example event would be the creation of model fragments by type. The VCs can build experience values depending on the user activities and his/her user model attributes this way.
- Implementation of a "Tell me what you know" functionality that reflects the current knowledge of the avatar requires treatment of the CM and submission to the VC to present this information. According request and answer communication is implemented between CM and VC.

5.2.4.2. API

Data structures shared with the CM component contain:

- Key Event Notification (WB/CM → CV), i.e. created new model
  - Parameter: Event describing structure, i.e. event id
- Knowledge Request (CM ← VC), i.e. "Tell me what you know" user request
- Knowledge Request Answer (CM → VC)
  - Parameter: Actual content representation (both model and simulation) to be displayed

5.3. Virtual characters design – content (phase 2)

This section describes the characters due by project M30.

5.3.1. Critic

Critic given by a virtual character occurs in two different use-cases. One involves presenting the results of the diagnosis and assessment of a model (Section 3) to the learner in an appropriate way. The other is the likewise presentation of ontology-based quality feedback (Section 4).

Note that the character giving the critic could have two different appearances and mannerisms depending on what is being presented. For example, when giving quality feedback, the character could appear very strict and unforgiving (a real critic) while he could be friendly and helpful (a caring teacher) when diagnosing the learner's model. Figure 5.5 shows the architectural design for quality feedback, Figure 5.6 that for diagnosis and repair.

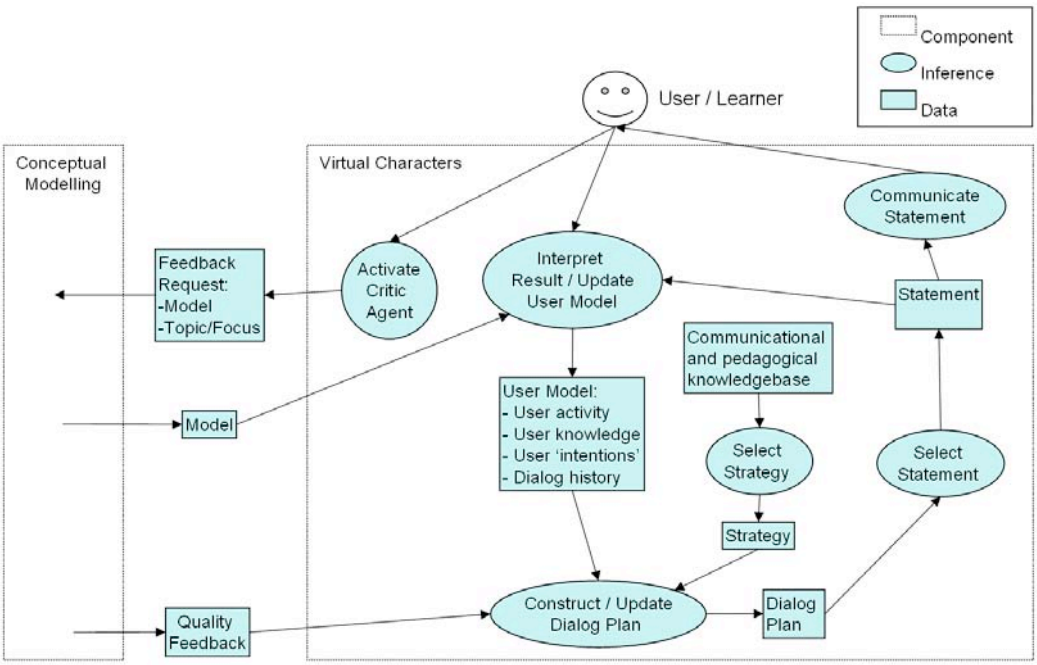

Figure 5.5: Architectural design for a virtual character giving quality feedback.

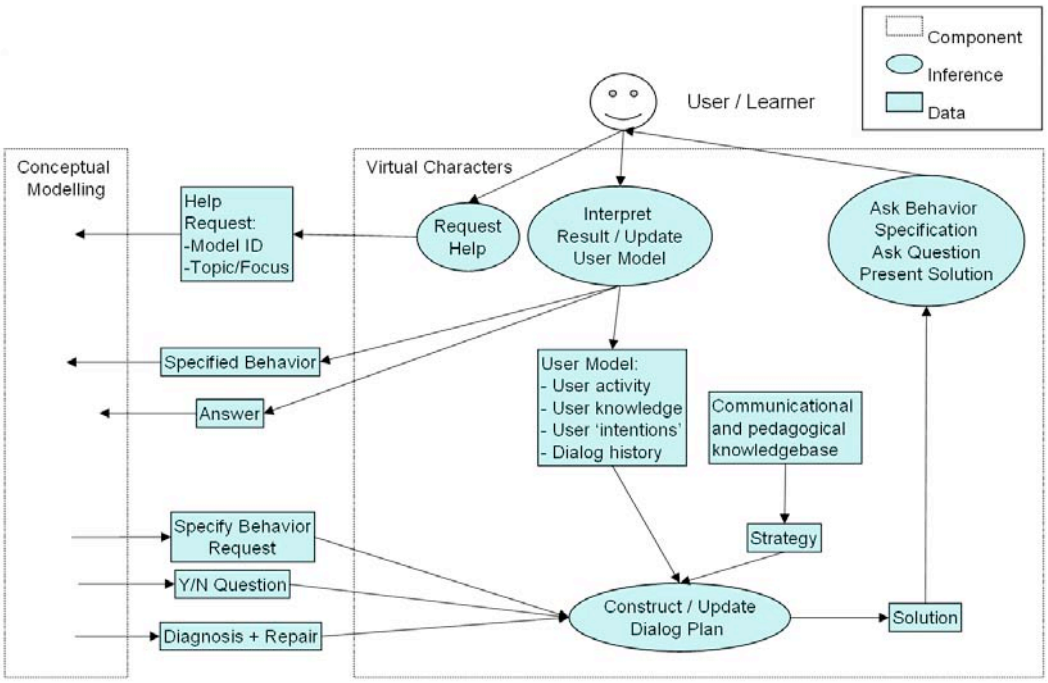

Figure 5.6: Architectural design for a virtual character presenting diagnosis and repair.

5.3.1.1. Components

As shown in Figure 5.5, the following components take part in a virtual character presenting quality feedback:

- The Learner indicates his wish to consult the critic, for example by pressing a corresponding button. A request is then sent from the VCs to CM, indicating the model that is to be criticized, as well as an optional constraint, such as a focus (only a certain part of the model) or a topic (only quantities for example).
- CM sends back both a representation of the current model and quality feedback. The representation of the model is employed to keep the user model up-to-date.
- Based on the user model and a communicative strategy selected from the pedagogical knowledge base, the quality feedback is transformed into statements, which the character presents to the learner.
- Based on the statements, the dialogue history within the user model is updated to avoid the repetition of equal or similar statements in future dialogs with the learner.
- Also, the learner might be given the opportunity to answer or react to what the character tells. This answer will also be used to update the user model.

The architecture in Figure 5.6 functions in a similar, yet more involved manner:

- Similar as above, the learner activates the agent resulting in a request to CM. The result of this request is a representation of the diagnosis and repair guidelines. Based on the user model and the pedagogical knowledge base, a proper dialogue is delivered by the virtual character.
- However, it could occur that the diagnosis cannot be completed without further information from the learner. This request for further information is sent by CM and is

again presented through the agent in form of a dialog. The learner could be prompted to answer a yes or no question or to specify the desired model behaviour. Based on the learner's response the diagnosis of the model can then continue.

5.3.1.2. API

For giving quality feedback, data structures shared with CM contain:

- Request for Feedback (CM ← VC)
  - Parameters: ID of desired model, enumeration value representing desired topic / focus
- Answer to Feedback Request, Model (CM → VC)
  - Parameter: Formal representation of model
- Answer to Feedback Request, Quality Feedback (CM → VC)
  - Parameter: Formal representation of quality feedback

In case of the agent presenting a diagnosis, data structures shared with CM contain:

- Request for Help (CM ← VC)
  - Parameters: ID of desired model, enumeration value representing desired topic / focus
- Answer to Help Request (CM → VC)
  - Parameters: Formal representations of diagnosis and repair guidelines
- Request for Specifying Behaviour (CM → VC)
  - Parameter: None
- Answer to Specify Behaviour Request (CM ← VC)
  - Parameter: Formal representation of desired behaviour
- Yes or No Question (CM → VC)
  - Parameter: Formal representation of question
- Answer to Question (CM ← VC)
  - Parameter: Yes or No

5.3.2. Model comparison

Different variants of dialogues will be explored. For example, a virtual teacher could compare an expert model with a model created by a student, or two teachable agents could engage in a dialogue about their models. Figure 5.7 shows the architectural design for this.

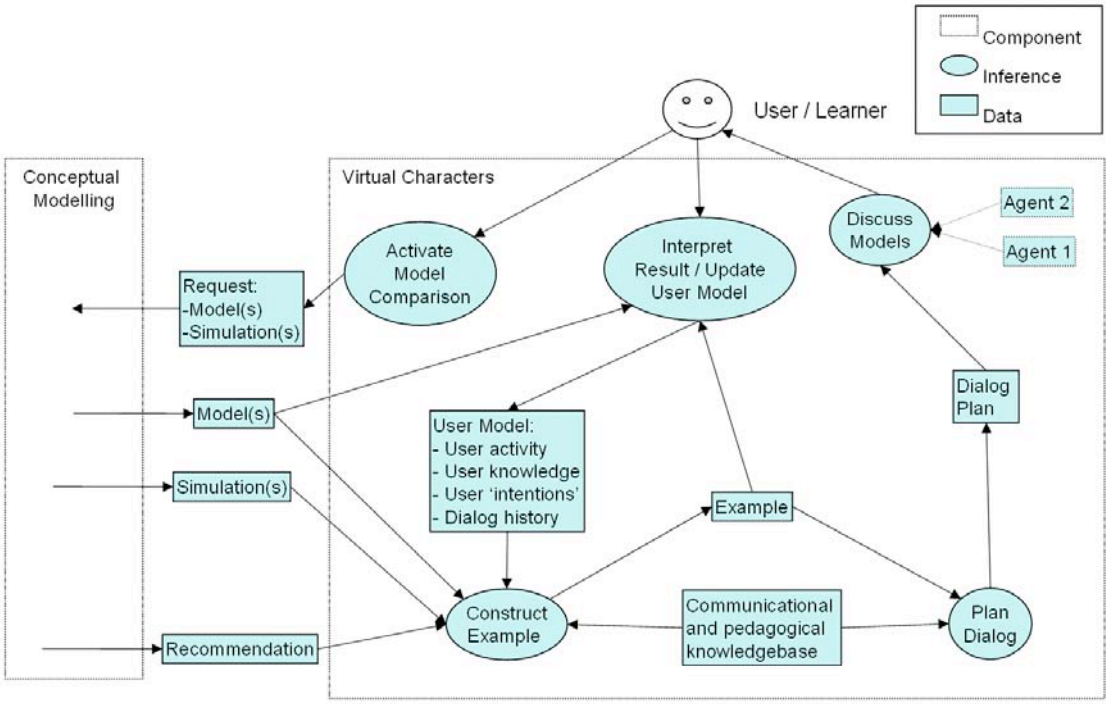

Figure 5.7: Architectural design for Model Comparison with virtual Characters.

5.3.2.1. Components

Model Comparison involves the following components:

- The Learner indicates his wish to compare his model to another one. A request is then sent from the VCs to CM, indicating the model(s) to be compared.
- CM sends back a representation of the model(s) as well as a representation of the simulation result(s). The representation of the learner's model is employed to keep the user model up-to-date.
- CM also sends back a recommendation for interesting differences between the two models. Based on this, the simulation result(s), the model(s), the user model and the pedagogical knowledge base, a dialogue is created that shows and explains these differences.
- The dialogue is then presented by the agent(s).
- The content of the dialogue as well as any user response to it are fed into the user model.

5.3.2.2. API

Data structures shared with CM contain:

- Request for Model Comparison (CM ← VC)
    - Parameter: ID(s) of desired model(s)
- Answer to Request, Model (CM → VC)

- Parameter: Formal representation of model(s)

- Answer to Request, Simulation (CM → VC)
  - Parameter: Formal representation of simulation result(s)
- Answer to Request, Recommendation (CM → VC)
  - Parameter: Formal representation of recommendation

5.3.3. Quiz

During a quiz, the quiz character asks the learner multiple-choice questions about the learner's model. The architecture for this is shown in Figure 5.8.

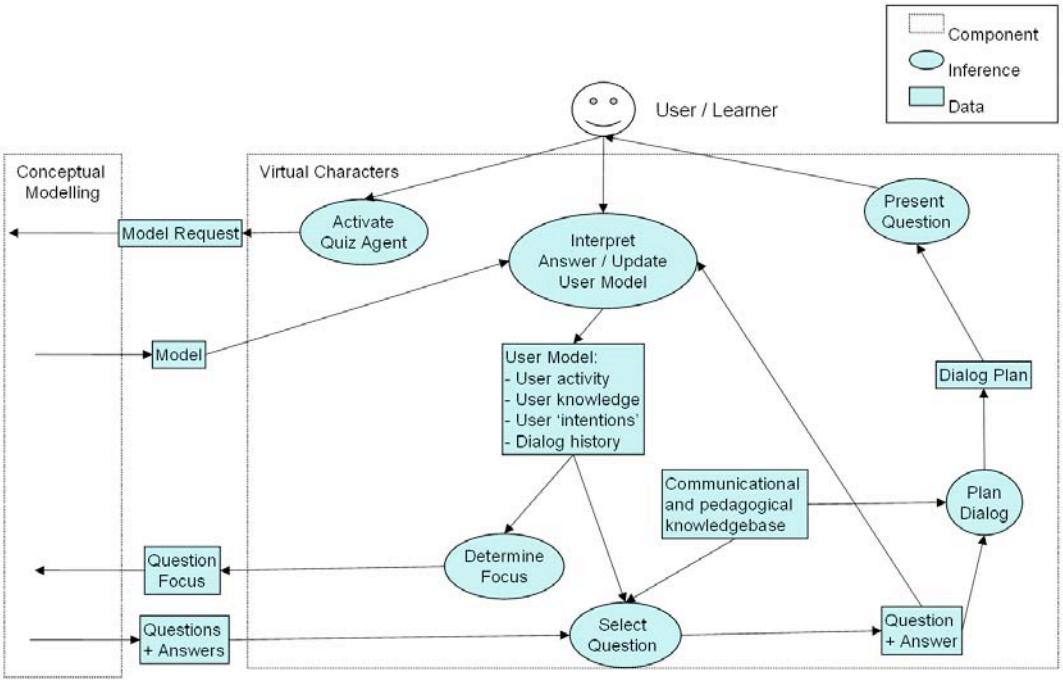

Figure 5.8: Architectural design for the Quiz virtual character.

5.3.3.1. Components

During a Quiz, the following components are involved:

- The Learner indicates that he likes to take a quiz. A request is then sent from the VCs to CM, indicating the model upon which the quiz will be based.
- CM sends back a representation of the model, which is used to update the user model.
- Based on the user model, the focus of the questions is determined and sent to CM.
- CM sends back a list of questions, each with multiple answers as an indication of the right one. Based on the user model and the pedagogical knowledge base, a question is selected from this list. Next, a dialogue plan presenting this question along with its answers is generated.

- Finally, the character puts the question to the learner.
- The user model is updated with the question and the learner's answer to it.

5.3.3.2. API

Data structures shared with CM contain:

- Request for Quiz (CM ← VC)
  - Parameter: ID of desired model
- Answer to Request, Model (CM → VC)
  - Parameter: Formal representation of model
- Setting Question Focus (CM ← VC)
  - Parameter: Enumeration value representing desired focus
- List of Question and Answers (CM → VC)
  - Parameters: Formal representation of question and answers, indication which answers are correct

5.3.4. Learning companions – Chatting with other (human) learners

The learning companion presents the interaction with a virtual peer, such as chatting or showing one's model to each other. The architecture for this is shown in Figure 5.9.

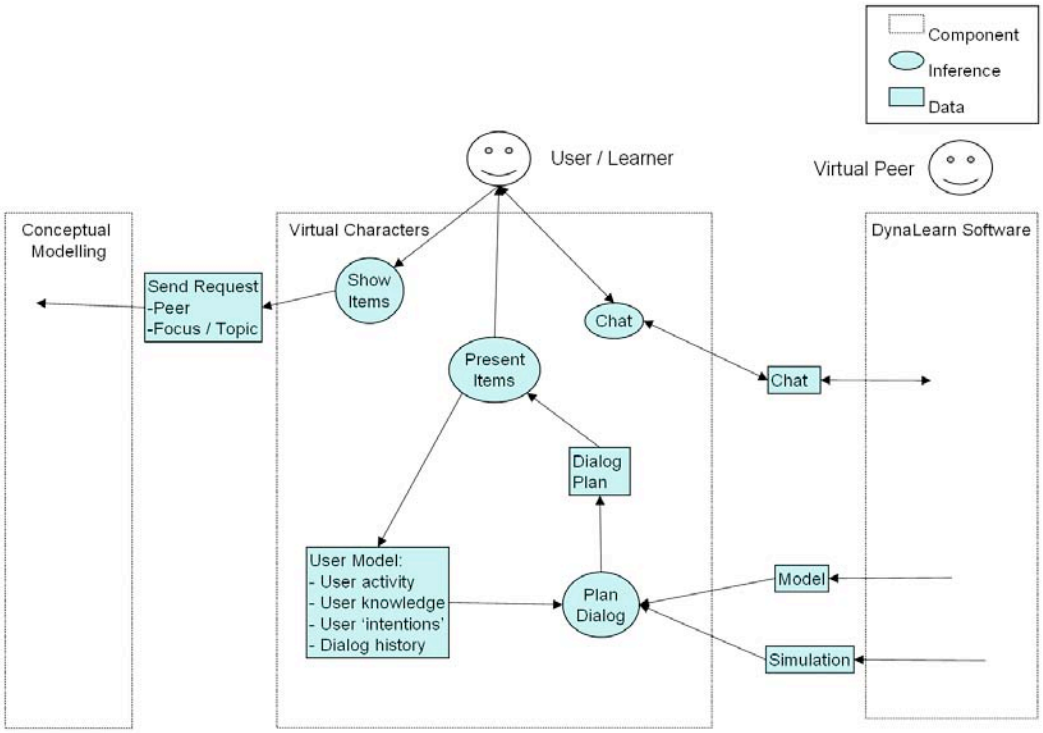

Figure 5.9: Architectural design for a virtual character acting as a learning companion.

5.3.4.1. Components

When interacting with the learning companion character, the following components are involved:

- Learners can chat with other learners via their respective virtual peers. In this case the learning companion character then directly presents the typed text.
- Learners can also show their model (or parts of it) to the other learners. A request is then sent to CM, indicating a focus or topic and the peer this information should be sent to.
- If the peer in turn sends information about their model or simulation details, a dialogue is constructed with help of the user model. The dialogue is then presented by the learning companion character.

5.3.4.2. API

Data structures shared with CM contain:

- Request to Send Model to Peer (CM ← VC)
    - Parameters: ID of desired model, ID of desired peer, enumeration value representing desired topic / focus
- Incoming Chat Message (CM?? → VC)
    - Parameter: Chat message in plain text
- Outgoing Chat Message (CM?? ← VC)
    - Parameter: Chat message in plain text
- Model / Simulation Results from Peer (CM?? → VC)
    - Parameters: Formal representation of model / simulation results

The exact organisation and inclusion of the virtual link with peers requires further exploration, particular in the context of the conceptual modelling workbench and its appearance after M18.

5.4. Multilingualism

Making the VC multilingual or adding a new language respectively, would require changes both to the dialogue / text and to the voice generation:

- Dialogue / Text generation: Content determination and structuring of a dialogue are language-independent and thus can be used across different languages. However, lexicalization and structure realisation are not only language-dependant but their complexity greatly varies with different languages. For example, due to morphology, creating dialogs in German is much more difficult than in English. DynaLearn will support English. Pre-scripted parts, such as pre-stored presentations of characters may be provided for other languages as well since these may be authored by partners from the respective countries.
- Voice generation: There are a few free Text-to-Speech (TTS) systems available, covering different languages and with varying range of quality. Further research and

fine-tuning is required to establish the project needs (e.g. coming from WP7) and what can be realised given the available technology and project resources.

The question of multilingualism is also a question of whether the DynaLearn software will support Unicode, e.g. for Portuguese and German special characters (such as umlauts).

5.5. Planning virtual character development

Figure 5.10 shows the planning for the virtual character development. To accommodate yearly reviews and evaluation requests form WP7, prototypes of the most achievable virtual characters are scheduled for M12 and M24.

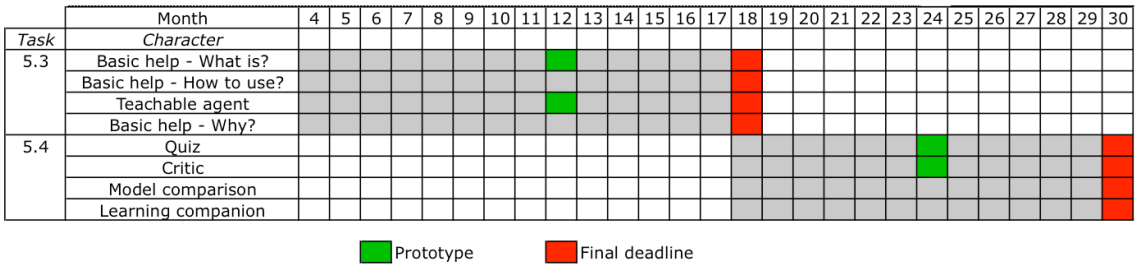

Figure 5.10: Planning virtual character development.

The realisation of the character depends on the CM and ST development. Particular dependencies on other tasks include:

- The Basic Help characters depend on Task 3.1 “Multi use-level workbench”, running from month 4 to 10.
- The Quizmaster character depends on Task 3.3 “Question generation and answering” which runs from month 4 to 18.
- The Critic character depends on Task 3.4 “Diagnosis and assessment” running from month 7 to 30, and on Task 4.2 “Ontology based feedback on model quality” running from month 12 to 18.
- The Model Comparison characters depend on Task 4.3, which runs from month 19 to 30.
- The Learning Companion character depends on the ST (WP4).

5.6. Architecture virtual characters design component

Virtual Characters are an enriching addition for a software product from a functional and user-oriented point of view. This role as an enhancement to a fundamental main component is also picked up in the technical arrangement of the DynaLearn component composition. The diagram in Figure 5.11 displays a schematic description of the main roles that participate in the architecture concerning the integration of virtual avatars into the DynaLearn software project.

The task of the VC Component is to present content provided by GARP in an appropriate manner. This includes the animation of communicative behaviours and their synchronization with speech bubbles (or speech). The plug-in characteristic emerging from this concept makes it easy to switch between 2D and 3D display of the same source data with low effort provided that each pluggable client supports the same protocol command sets. Ideally no changes on GARP side are required when another client is plugged into the component arrangement. This is

especially helpful when supporting different visualizers, such as a frame-based animation approach and a real-time 3D animation engine in parallel.

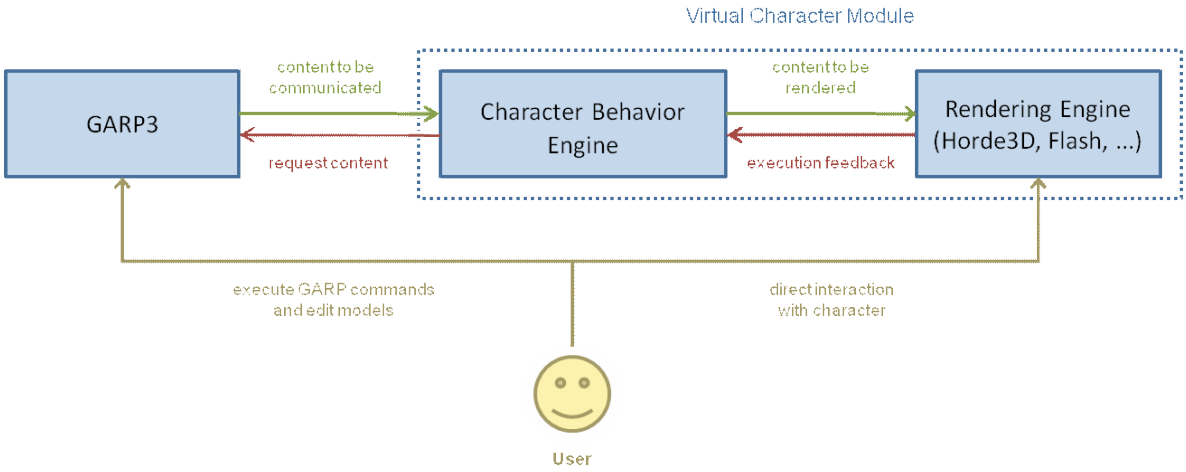

Figure 5.11: Basic architecture of the communication data exchange between the virtual characters and another software component like GARP. The connection is realized using sockets and an XML data exchange format.

The following list summarizes the components and their functionalities:

- ***GARP3*** sends contents to be communicated to the VC Component for further processing. During the interaction with the VC Component, GARP3 may also receive explicit requests for specific content delivery.

- The ***Character Behaviour Engine*** decides how content provided by GARP3 should be communicated to the user using gestures, mimics and speech bubbles (or speech). It transforms contents received from GARP into rendering instructions and sends them to the Rendering Engine. The rendering instructions may include dramaturgical actions, such as moving a character to a particular position on the screen or in a 3D environment, as well as multimodal speech acts, such as asking or answering a question. In the latter case, natural language text messages are built using a template-based approach and provided along with character behaviours, such as head nods. Because textual explanations are built directly out of the model structures it is most efficient to write the dialogue builder as a Prolog module so that data can be accessed directly without long ways round. A first prototypical test for that module has been developed. Vice versa, the Character Behaviour Engine also receives feedback from the renderer, such as execution states or animation progress.

- The *Character Rendering Engine* is responsible for the audio-visual rendering and control of the characters to be shown on the screen. Because of an abstract design, the rendering engine is easy replaceable so that different implementations can be developed with low effort. Currently, we support two rendering engines: the Horde3D GameEngine for real-time 3D animations and a frame-based Flash engine that has been developed by UAU as well.

- The ***User*** interacts with the GARP editing environment to build models and contents. In addition, he or she may also interact with the VC Component directly, for example, via a menu-based interface or a text box. An internal message is then generated and sent through the behaviour engine module, which may perform additional interpretations or conversations of the received signals before the message is routed to GARP.

5.6.1. XML protocol for data exchange

The commands sent to the VC components are phrased using an XML format which is easily readable by humans. Development and debugging tasks become easier in consequence. Furthermore handling and processing of screenplay scripts is possible for nearly any software component, programming environment and operating system because of the simple text nature of XML. Because also line breaks are possible contents of xml documents they cannot be used as segment dividers. Each complete xml document is therefore separated through a final null byte when the streams are sent over the sockets.

The draft of the XML format describes a screenplay containing a simple list of commands. These instructions can be arranged in two categories. *Descriptive* commands affect the management of objects in a slightly technical manner and may include actions like the creation of avatar instances or the change of properties like visibility. The second category contains *conductive* instructions that directly concern dramaturgy and action flow on screen. Conductive examples contain the vocalization of talks or playing specific entertaining animations.

Screenplay scripts for the virtual characters are intended to be content-oriented and technically independent. This means that the xml element scope was designed with the aim to give the script editor the possibility to concentrate on contents and not to technical specifics. Therefore screenplay scripts should specify the general staging of avatars and dialogue sequences which depend on the main software's current context and contents. Also specific animations can be controlled through the xml command sets as long as they refer to actual dramaturgical actions. By contrast, transitional animations that only round the visual experience should be done implicitly by the VC Component. For example a cheering animation could be triggered through script as reaction for a successful user event in the software. The cartoon directly visualizes the achievement and can be controlled explicitly in this case. When the character sits on a chair and starts walking the motion of standing up has not to be controlled via screenplay scripts because it only bridges between sit and walk states and has no direct content relation. Another design target for the screenplay xml format was that script editors should not have to care technical specifics like 3D engine render flags for example. This is especially important for the earlier mentioned plug and play concept that allows the integration and replacement of different visualizers such as 2D or 3D engines without affecting the client side interfaces.

The implementation of the xml script specification follows an incremental stack model which means that multiple sent screenplay xml documents do not replace each other, but merge into a single command list. This ensures that running processes like animation sequences or running dialogs can still quit as needed and are not aborted abruptly.

An example data exchange flow using screenplay scripts and xml answers is shown in Figure 5.12. Granted that the sequence occurs in an unaffected startup situation without precedent inputs or events the display of a virtual character that asks a multiple choice question and sends the user's answer back to the main software could happen the following way. First the main application creates and submits a screenplay document that causes the VCM client to create a new avatar and display it on the screen (A). The following "set" request (B) activates the quiz mode of the character. It is possible to integrate this command into the earlier screenplay block but in later practice most likely the request will occur with already present avatars so that initial scripts (A) like the one in this example are not necessary.

The VC confirms the successful change mode switch by sending back an answer (C). The request of a new question is triggered here immediately although it can be repeated again at any other points in time to generate a new challenge. With the question block (D) the server reacts on the request and creates a question out of the model knowledge that is sent back with a multiple answers. One of the answers is marked as the correct one using the solution type

attribute. After the client side interpretation of the question and exposure in the graphical interface the user is able to pick a question solution. This answer is finally sent back as a value pair of question ID and correctness of the answer (E).

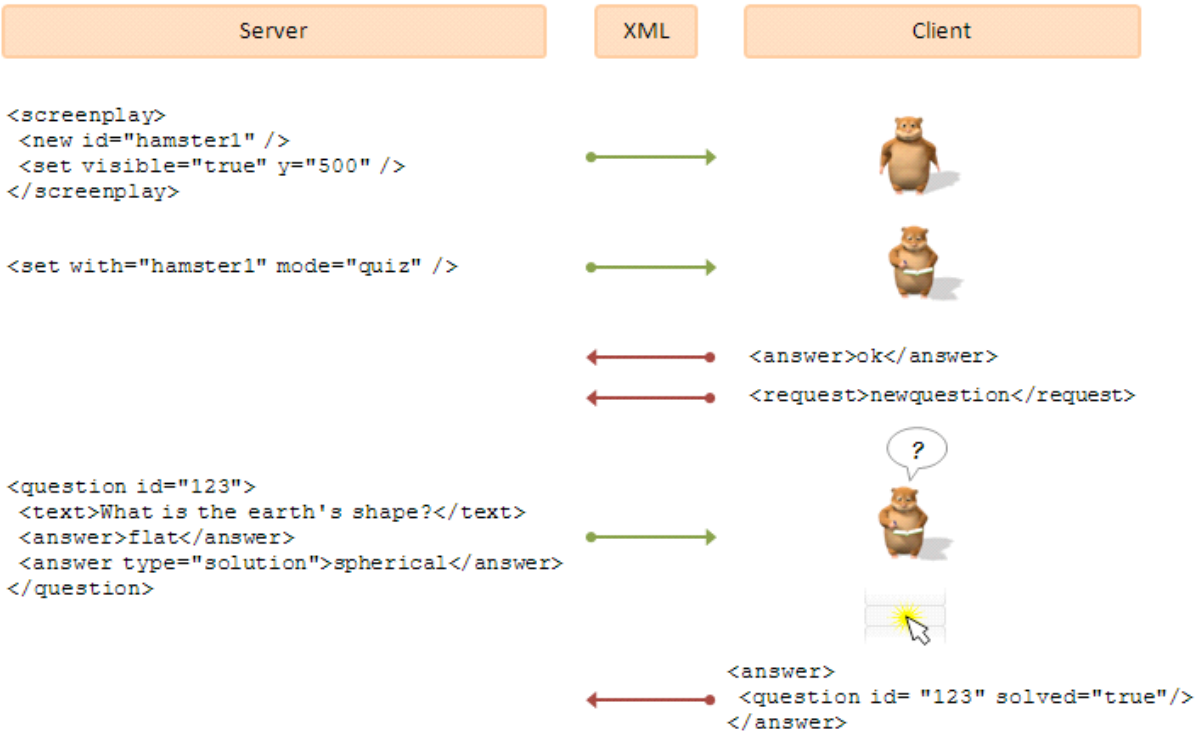

Figure 5.12: Performing a multiple choice question quiz as example for the flow of xml screenplays between a serving script source and the VC component.

5.6.2. Proof of prototype concept

5.6.2.1. Window integration of avatars

During the technical planning of the Avatar concepts for the DynaLearn project the question came up of how to implement platform independent 3D characters that fit into the GARP3 editing environment for a consistent user experience. Beside Microsoft Agent which is quite far developed concerning virtual agents tasks but limited to the Windows platform, Adobe Flash came into focus especially with its youngest trends towards rich desktop applications beyond web browser windows.

DynaLearn occurs as a consolidation of multiple software components developed by the participants so that avatars need to be integrated in external third party software products like GARP3. Therefore, it is hardly possible to create a complete entertaining learning environment which would require that all graphical contents be integrated into the 3D scenery. Figure 5.13 displays some examples for possible integration approaches for virtual avatars. Opening two parallel windows (A) would interrupt the immersive user experience highly, because the authoring environment and the avatar's living space would be recognized as a separate component with perceptible break. The character is captured here inside the window and interaction with outside elements is not possible. Furthermore the character's window allocates screen space and obtains the meaning of a video window. The helper experience for the user is

comparable with a phone call to a friend. It is difficult to implement multiple avatars a time because the window would need to be resized which leads to an even bigger occlusion of workspace. This problem is even worse in the embedded window (B) as used in “Betty’s Brain” [11] for example where no window extension is possible. All avatars would have to share the small space available.

For an improved integration of the avatar into the learning environment formless overlay animation should be used like in the case of Microsoft Agents (C). The figure is then perceived as element of the complete desktop where it can move freely. It is possible to walk or fly to any location on the screen when explanations of specific screen elements are requested. Even for lightweight software components it should be quite easy to translate local coordinates of the canvas to screen coordinates that can be sent to the agent system.

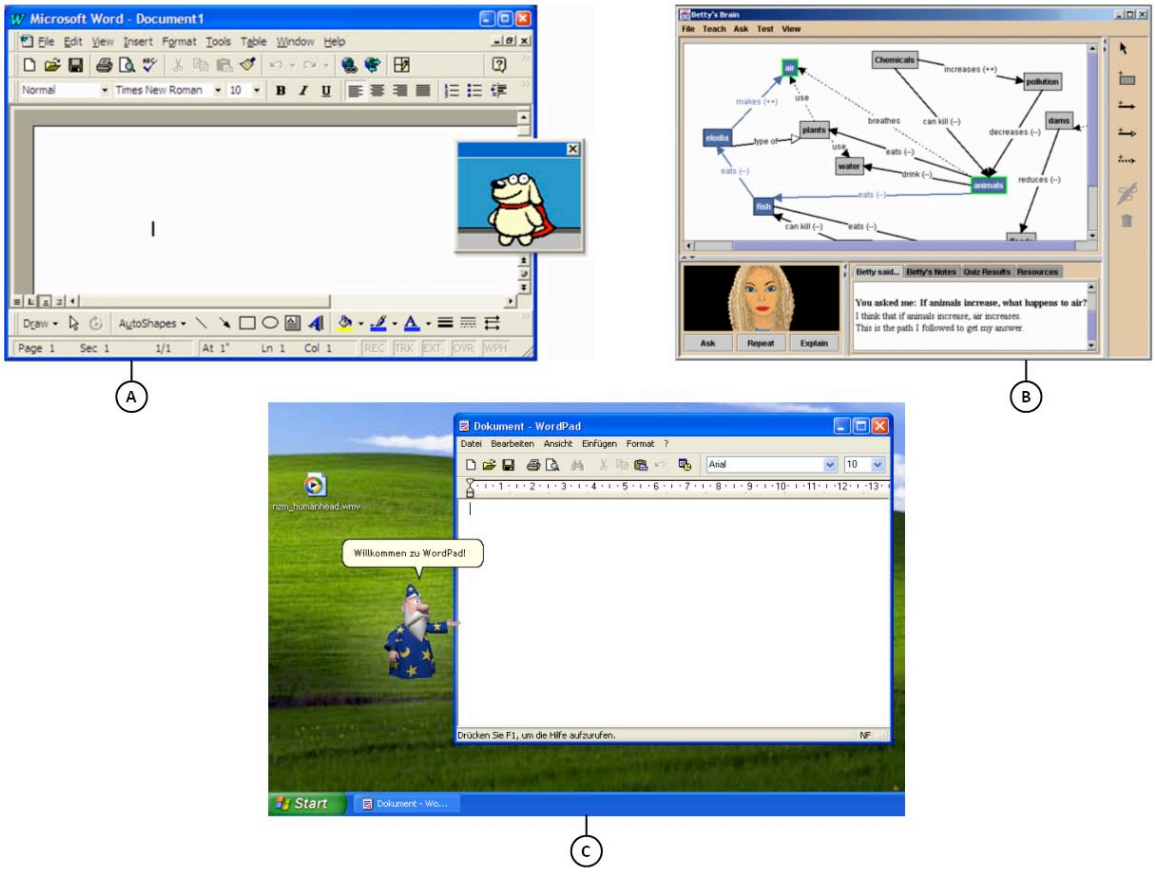

Figure 5.13: Approaches of integrated avatars with floating, integrated and free form windows.

5.6.2.2. Main requirements

The following technical main requirements emerge from the technical architecture for the VC Component:

- High *Modularity* is aimed by the creation of an encapsulated easy to handle component
- Communication interfaces should be implemented via *Sockets and XML*
- The whole software product has to be *Platform Independent*

- For comfortable usage an *Easy Install Process and Few Runtime Dependencies* are preferred
- Images and animation ideally integrate visually seamless using *Frameless Overlays*

Two main technologies where found by UAU to be suitable for the implementation of the avatars: Adobe's Flash/Flex/Air frameworks and the Horde3D GameEngine. The following Table 5.1 compares some characteristics, which are relevant for the applicability.

Table 5.1: Comparison of technical aspects for the Flash and Horde3D based character client approaches.

	Adobe Flash	**Horde 3D GameEngine**
Modularity / Distribution	Self-containing movie clips	Zip files and file collections
Required runtime frameworks	Flash Player and Air runtime	GameEngine executable
Socket support	Usage of factory-provide classes	Usage of existing classes and functions
XML support	Native access via E4X scripting	Usage of existing XML modules
Platform independence	Platform runtimes available	Given by engine design
Programming Language	Action Script Language	C++
Avatar presentation	Frameless overlay window	Window or frame with 3D scenery
3D capabilities	Rendered movie clips	Full hardware accelerated real time 3D
Runtime License	Free of charge usage	Open Source license

5.6.2.3. Flash client

One well-established technology for interactive multimedia applications is Adobe's Flash machinery. The modules generated by this grown-up framework are distributed as single files containing all necessary resources, which make handling and dissemination of contents very easy for users.

In the last decades the Flash technology experienced some big steps in product evolution and market success. This led to a widespread distribution of the Flash Player web browser plug-in that reaches up to 99% of today's Internet users according to the developer company [10]. Along with this extension the scripting language and the functional range of the development kit increased tremendously. While XML is already used for various data exchange concerns for years, more and more XML based utilities are integrated to expand the technical potential of Flash's script language Action Script. Beside the full support of the ECMAScript for XML (E4X) standard the factory-provided XML socket infrastructure is an example.

The improved quality and quantity of possibilities in Flash development also enlarge the potential fields of application. Therefore the product is no longer reduced to work inside a web browser window, which is the classical use of Flash creations. Adobe's Air framework makes it possible to create standalone software products instead that can be executed like any other desktop application. As a matter of course Air applications are platform independent like any other Flash contents and very easy to install. Each platform specific task is managed

automatically by the runtime environment, which has to be only installed once on each operating system to make any Air application work.

Leaving the bounds of the web browser window decreases spatial restrictions and gives more freedom to content developers. As in other programming environments it is now also possible for Action Script developers to decide whether an application should use native windows, the lightweight GUI library or even free form windows. Latter can be used to place characters freely on screen and to maximize the integration experience for the DynaLearn main software. Early experiments confirmed the suitability of this approach as shown in Figure 5.14.

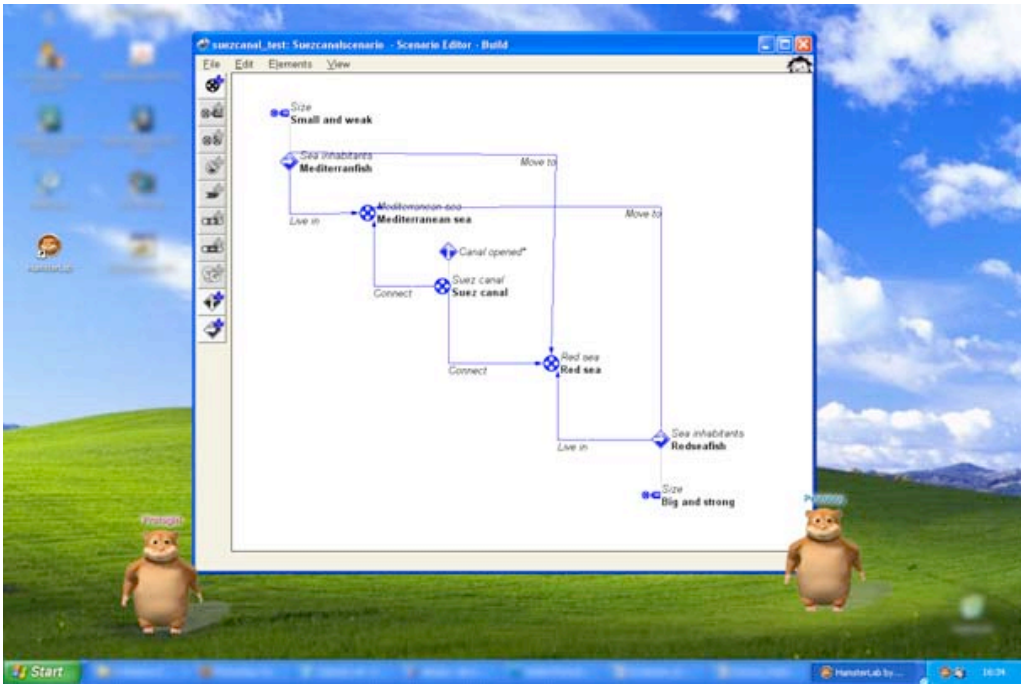

Figure 5.14: Virtual characters appear on the desktop without container windows using Adobe Air technology.

Another advantage of the freely available Flex and Air compiler SDKs is the possibility to make use of the 2D motion graphics based nature of Flash and its timeline animation features. These mechanisms support the effortless mixture of vector graphics, pixel based images like photos, sound effects, previously created or recorded movie footage and (rudimentary) real time 3D renderings. This is an essential prerequisite for the consolidation of material from different authors into complex platform independent multimedia applications. The Action Script 3 language provides moreover a comprehensive object oriented scripting language that is easy but also powerful to efficiently express interaction and flows in a content focused way.

5.6.2.4. Horde3D

Although some open source libraries already exists that allow the implementation of real time 3D rendering using Action Script code, true GPU accelerated 3D is not possible through Flash Player's native methods yet. The lack of hardware-supported rendering is one main disadvantage of the Flash approach, which makes it necessary to keep the polygon count of models very low and to reduce the scenery to least number of objects possible.

Real time 3D engines give the user more freedom in interaction because the graphics module is not bound to predefined movie sequences but able to render models and scenes freely with any customization, lighting or camera angle. Multiple animations for gestures and bodily

expressions can be merged to new complex motions. The possibility to change textures or equip models with stage properties and assets ease the production efforts and increase the amount of graphical varieties. Furthermore it is imaginable to use other game engine specific features like physics simulation to visualize modelled knowledge in a new and entertaining way. Examples for hardware supported rendering of high-resolution meshes, the simulation of physics and graphical effects like particle systems using the Horde3D GameEngine are shown in Figure 5.15.

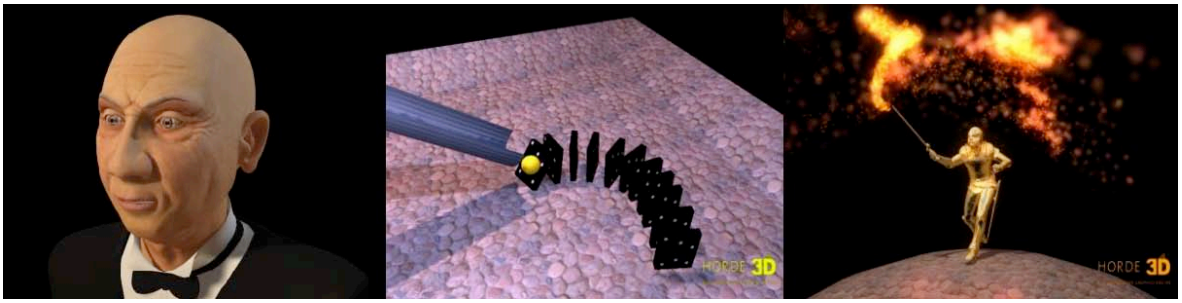

Figure 5.15: The Horde3D [12] real time 3D GameEngine allows complex mesh models, physics simulations and advanced effects like particle systems.

5.6.2.5. Test server

In order to get an easy to read knowledge base that allows a quick and easy start when testing the concept a simple program was created using the rule based Prolog language. This prototype took the role of the server in the investigational test-setup and allowed experimenting with question patterns. A template-based parser was developed which may be used for further language generation later on. The implemented socket connection for the exchange of the screenplay xml sequences could furthermore be integrated in other software tests towards the final product.

Screens from the prototype developed by UAU are shown in Figure 5.16. The developer is able to type xml screenplay code into a test window (A) of the server written in Prolog. Using the “Send” button this data is transmitted via a local socket connection to the VC prototype developed using Adobe Air. The developer console (B) of this application makes it easy to develop the components and to observe the data exchanges. After successful interpretation of the received screenplay script a first test character becomes visible on the screen (C).

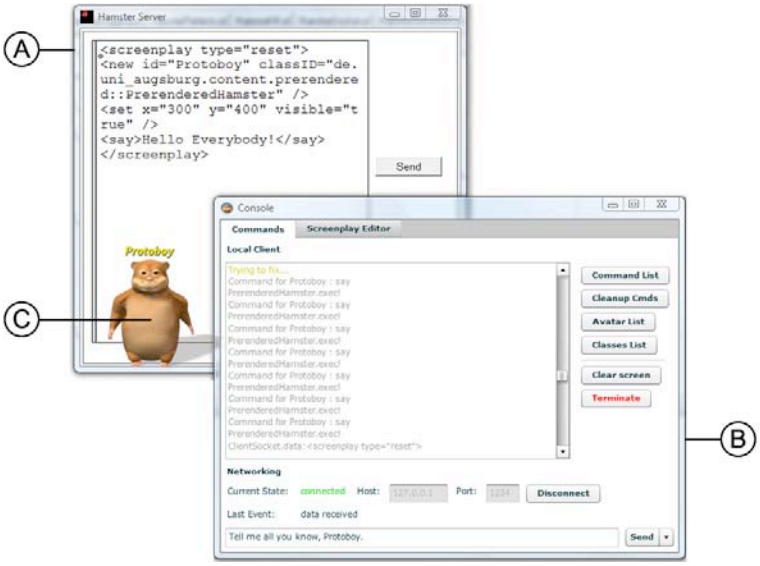

Figure 5.16: Experimental prototype for data exchange between Prolog software and VC.

5.6.3. Reflection on prototype

A technological architecture was sketched that provides a highly modular, flexible and easy to use composition of communicating software components. Socket connections are used as interfaces between the participants of the structural design to exchange information between the main application and the VC Component. The communication protocol follows an easy to read and write XML format that contains descriptive and conductive commands for remote controlling of the characters. Anyway this specification should only care about contents and dramaturgical elements while technically details should be handled as far as possible inside the character module.

Virtual Characters should not appear inside window frames to increase the integration experience for the user and to minimize the allocation of unnecessary screen space. During the evaluation for a platform independent and flexible multimedia authoring solution Adobe Flash was found as a powerful solution. In combination with the freely available Air SDK the bounds of the web browser can be left behind. As the developed prototype shows it is possible to create easy to use standalone applications with Air that implement frameless virtual characters directly on the desktop. A simple Prolog test server component also developed by UAU proofs furthermore the connection and data exchange capabilities between the components successfully.

The historically grounded limits of Flash concerning its real time 3D rendering potential leads to the idea of integrating UAU's powerful Horde3D GameEngine into the architecture. In general hardware-accelerated graphic engines have the advantage of a much greater flexibility when creating dynamic animations and customizations. Beside the easier generation of graphical design through combination of complex sub elements into a final result also new possibilities from the GameEngine area become accessible. An example would be the usage of a physics engine to visualize modelled phenomena in an intuitively understandable way.

Exchange of the 2D Air application and the real time Horde3D modules should be easy when the developed software follows the plug and play concept suggested in this document. In an ideal version no changes would be necessary on side of the script source (main application) when another character client is used.

6. Conclusion

This document describes the overall architecture of the DynaLearn interactive learning environment that allows learners to construct their conceptual system knowledge. It provides an overview of the Conceptual Modelling, Semantic Technology and Virtual Character components. The requirements from a learner perspective are translated to use-cases that describe the typical interaction with the tool. For each of these use-cases architectures and APIs have been developed that describe what data and knowledge has to be communicated between the different components.

As such, the document describes the API for the knowledge exchange between the CM and ST component, between the CM and VC component, and detailed requirements of the kind of virtual characters needed in the form of use-cases (i.e. the role they play). In addition a design of the user interface of the interactive learning environment is presented. Overall, this document formalises the consensus between the partners about the tool that will be developed and the way it will be implemented.

7. Discussion

The work described in this document is envisioned to be doable in the allotted time given the planning. However, some minor refinements are anticipated in the planning given the requirements of the experiments designed by the domain experts (i.e. giving certain features more priority). Furthermore, some refinements of the communication and the API are anticipated. For example, the Semantic Technology will probably not be accessed via direct socket connection, but be refined into a web-service that makes the service accessible to a wider audience. Furthermore, given the potentially large size of the OWL files (describing models and simulation results), we might choose to compress their content or make selections of which parts of the model and simulation should be saved. When working on the particular use-cases in each work package the coming years, the architecture, API and knowledge communicated will be further refined.

References

[1] Bredeweg, B., Salles, P. (2009). Mediating conceptual knowledge using qualitative reasoning, In: Jørgensen, S.V., Chon, T-S., Recknagel, F.A. (eds.), Handbook of Ecological Modelling and Informatics (pp 351 - 398) Wit Press, Southampton, UK.

[2] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. OWL web ontology language reference. W3C recommendation, World Wide Web Consortium (W3C), February 2004.

[3] Description of work (DoW). (2008) DynaLearn - Engaging and informed tools for learning conceptual system knowledge, Collaborative project (STREP), Call identifier FP7-ICT-2007-3, Project number 231526, October 9th, 2008.

[4] Salles, P., Bredeweg, B., Araujo, S., and Neto, W. (2003). Qualitative models of interactions between two populations. *AI Communications*, 16(4), pp 291-308.

[5] Goddijn, F., Bouwer, A. and Bredeweg, B. (2003). Automatically Generating Tutoring Questions for Qualitative Simulations. *Proceedings of the 17th International workshop on Qualitative Reasoning*, P. Salles and B. Bredeweg (eds), pages 87-94, Brasilia, Brazil, August 20-22.

[6] Liem, J., and Bredeweg, B. 2006. Document Type Definition (DTD) for QR Model Fragments, Naturnet-Redime, STREP project co-funded by the European Commission within the Sixth Framework Programme (2002-2006), Project no. 004074, Project Deliverable Report D2.3.1.

[7] Graesser, A.C., Wiemer-Hastings, K., Wiemer-Hastings, P., Kreuz, R., & TRG (1999). AutoTutor: A simulation of a human tutor. *Journal of Cognitive Systems Research, 1*, 35-51

[8] Gebhard, P., Kipp, M., Klesen, M., Rist, T.: Authoring scenes for adaptive, interactive performances. In: Proc. of the 2nd Int. Joint Conference on Autonomous Agents and Multi-Agent Systems, ACM (2003).

[9] André, E., Rist, T., van Mulken, S., Klesen, M. and Baldes, S., *"The Automated Design of Believable Dialogues for Animated Presentation Teams"*, Embodied Conversational Agents , J. Cassell, S. Prevost, J. Sullivan, and E. Churchill, The MIT Press (2000).

[10] http://www.adobe.com/products/player_census/flashplayer (last visited June, 30th 2009).

[11] http://aaalab.stanford.edu/teachable_agents/ta_betty.html (last visited June, 30th 2009).

[12] http://mm-werkstatt.informatik.uni-augsburg.de/project_details.php?id=54 (last visited June, 30th 2009).

Appendix A – Physical design learning spaces

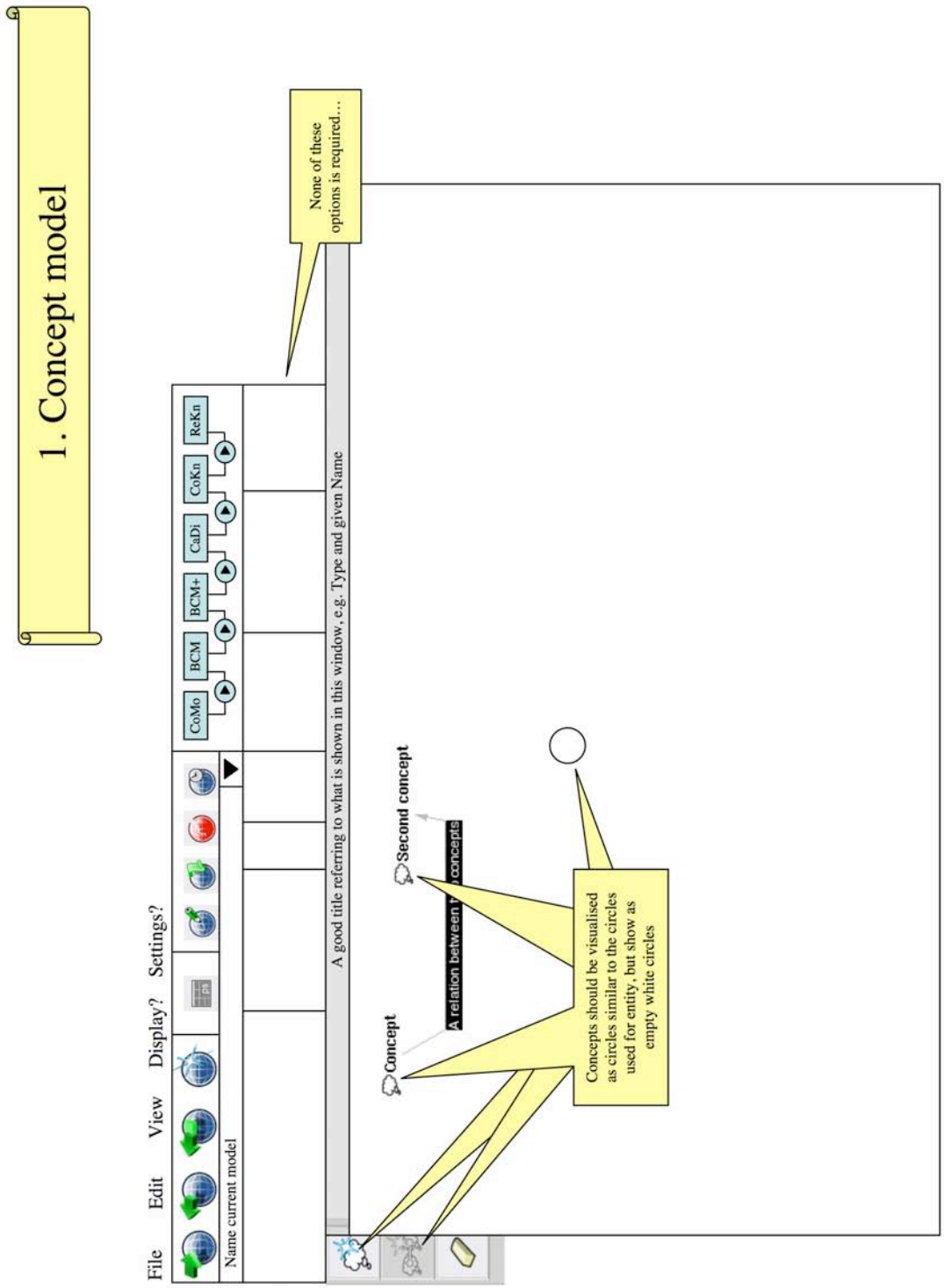

Figure A.1: Physical design Use-level 1 - Concept map.

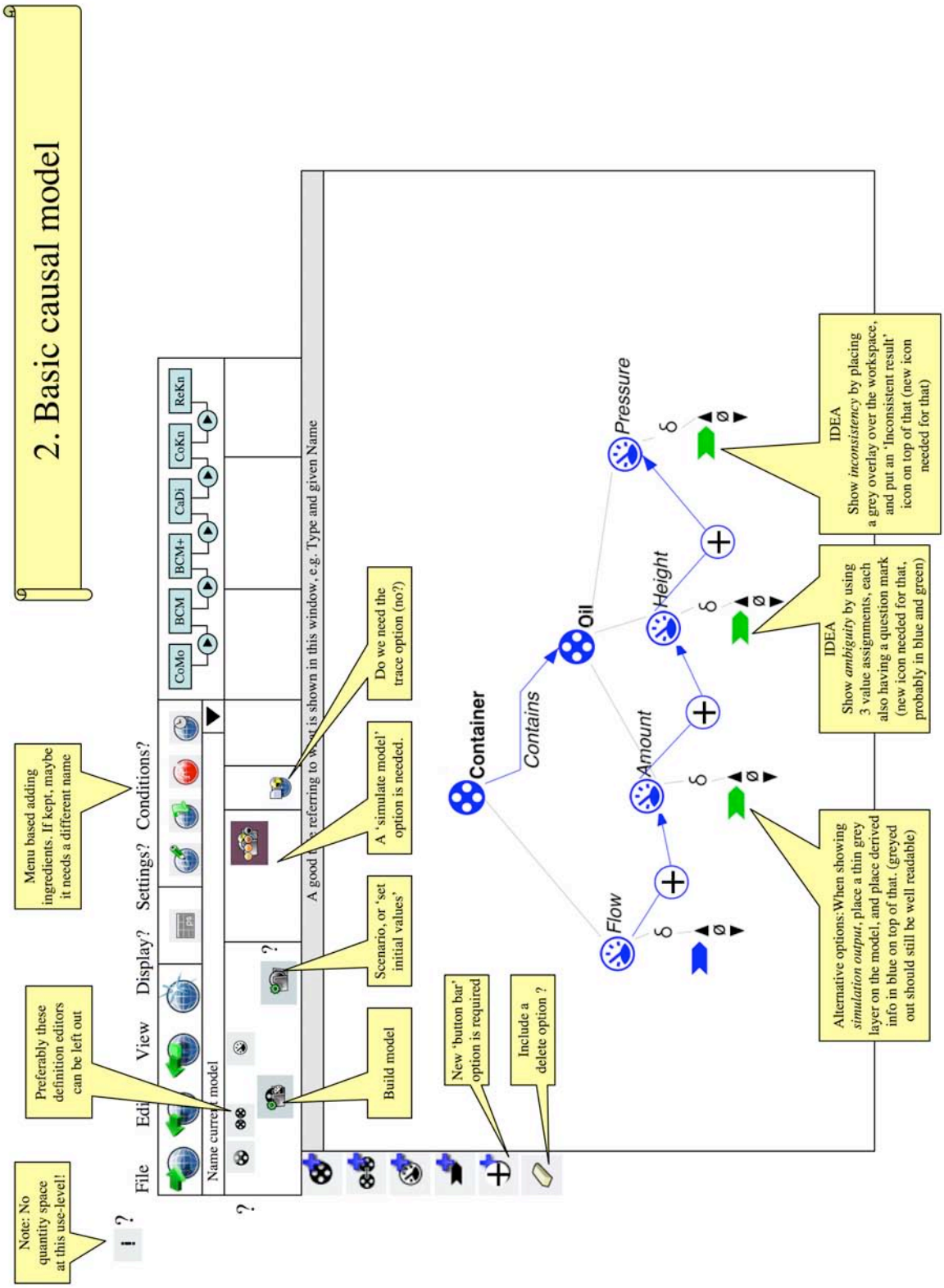

Figure A.2: Physical design Use-level 2 - Basic causal model.

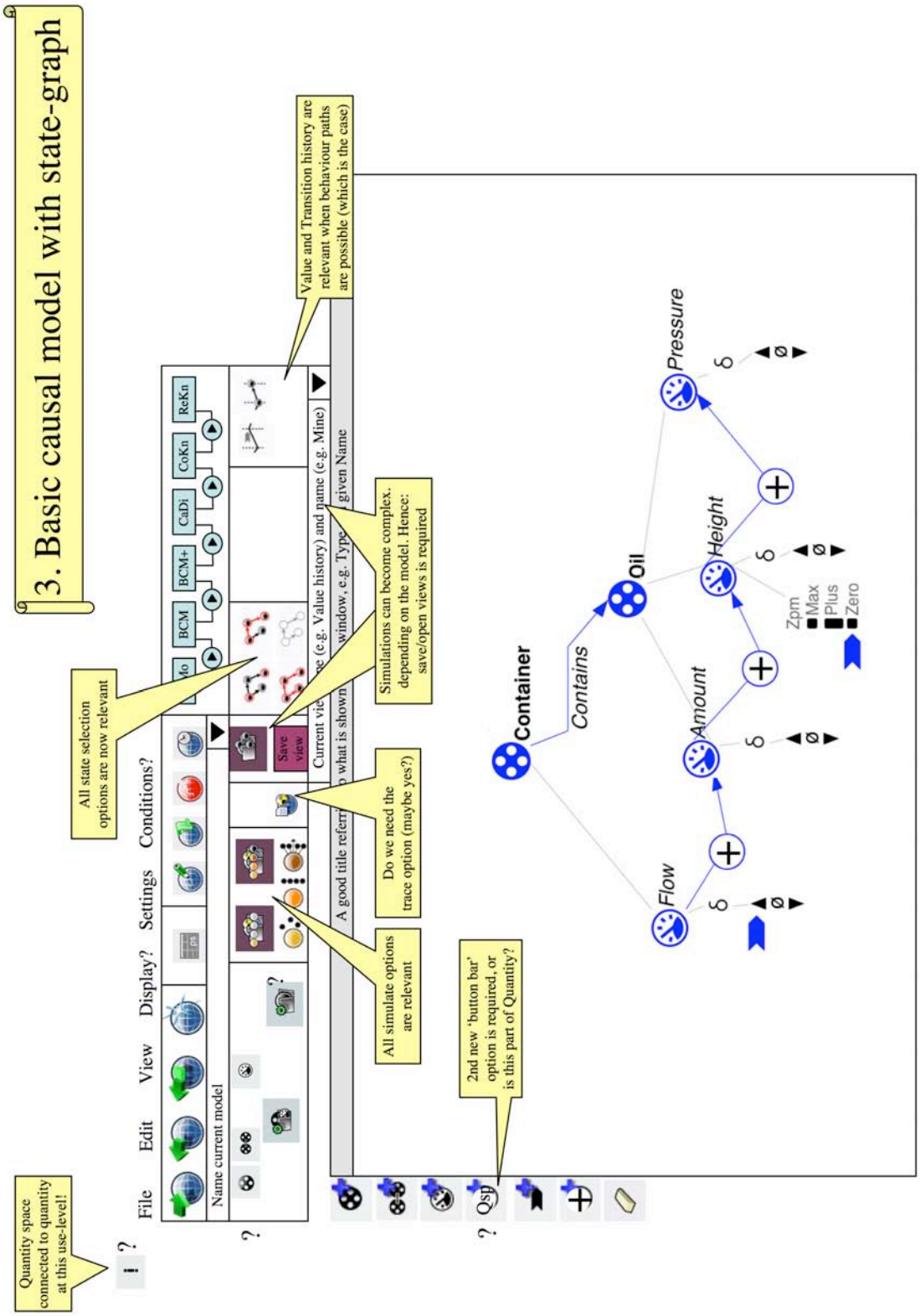

Figure A.3: Physical design Use-level 3 - Basic causal model with state-graph.

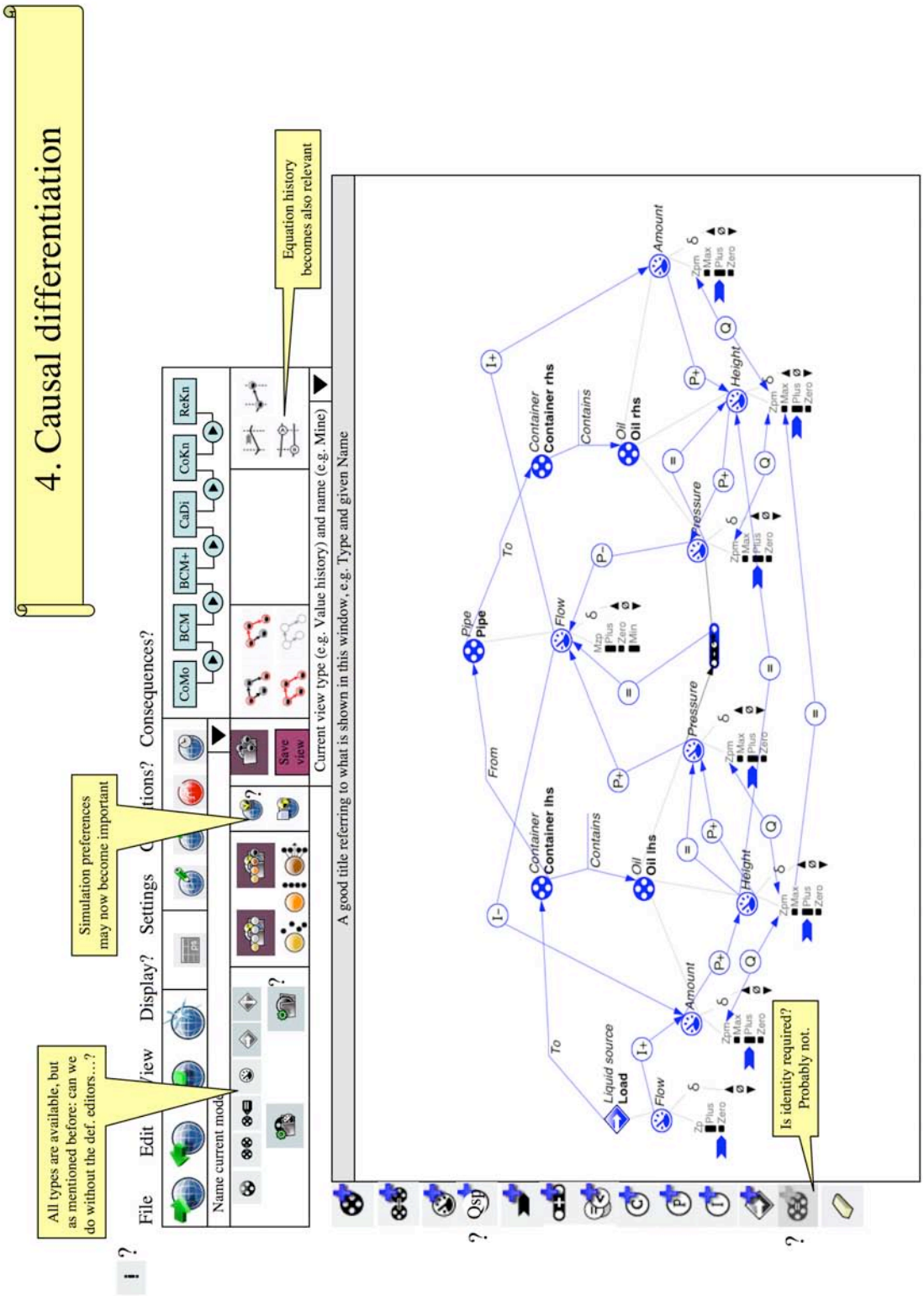

Figure A.4: Physical design Use-level 4 - Causal differentiation.

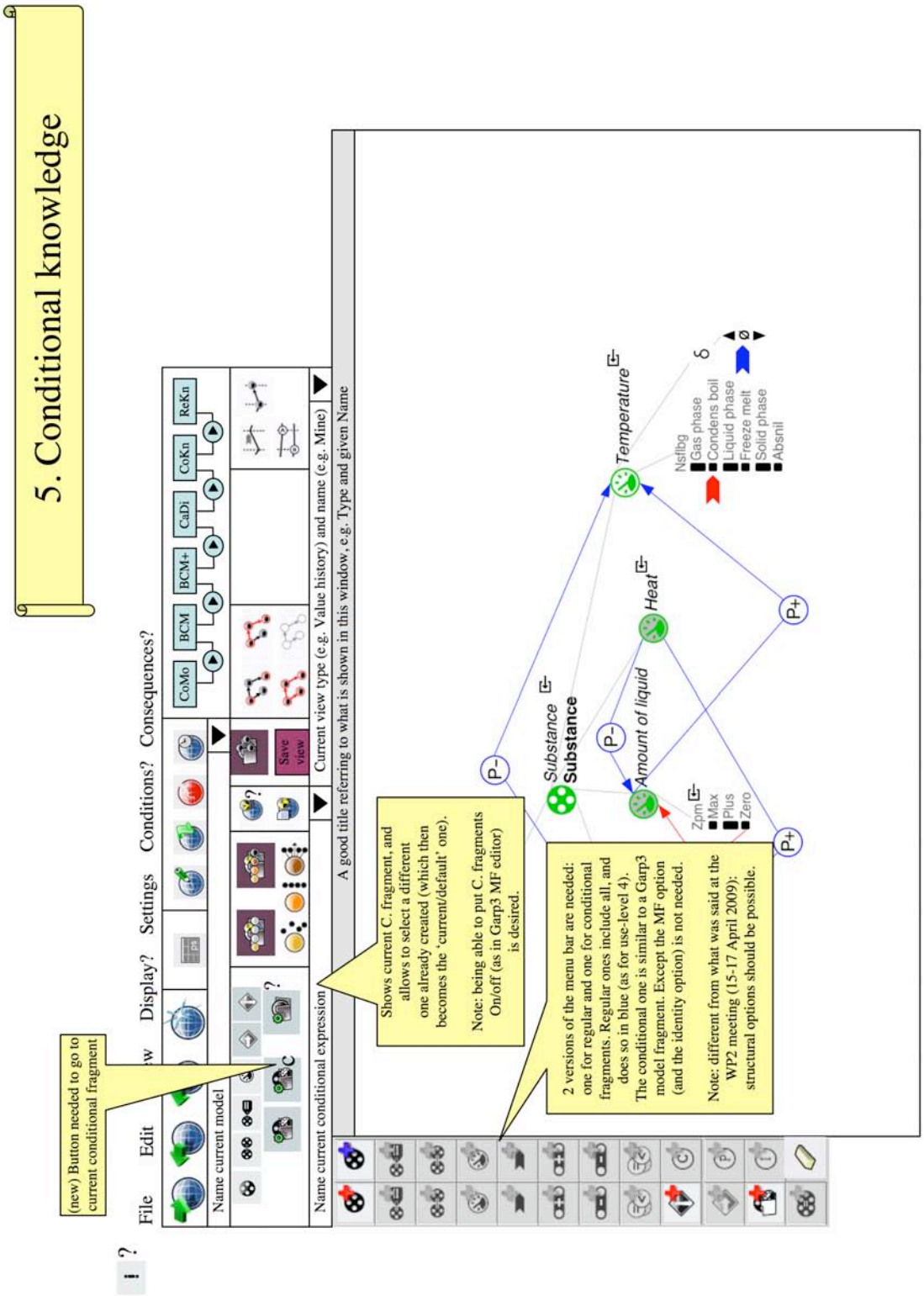

Figure A.5: Physical design Use-level 5 - Conditional knowledge.

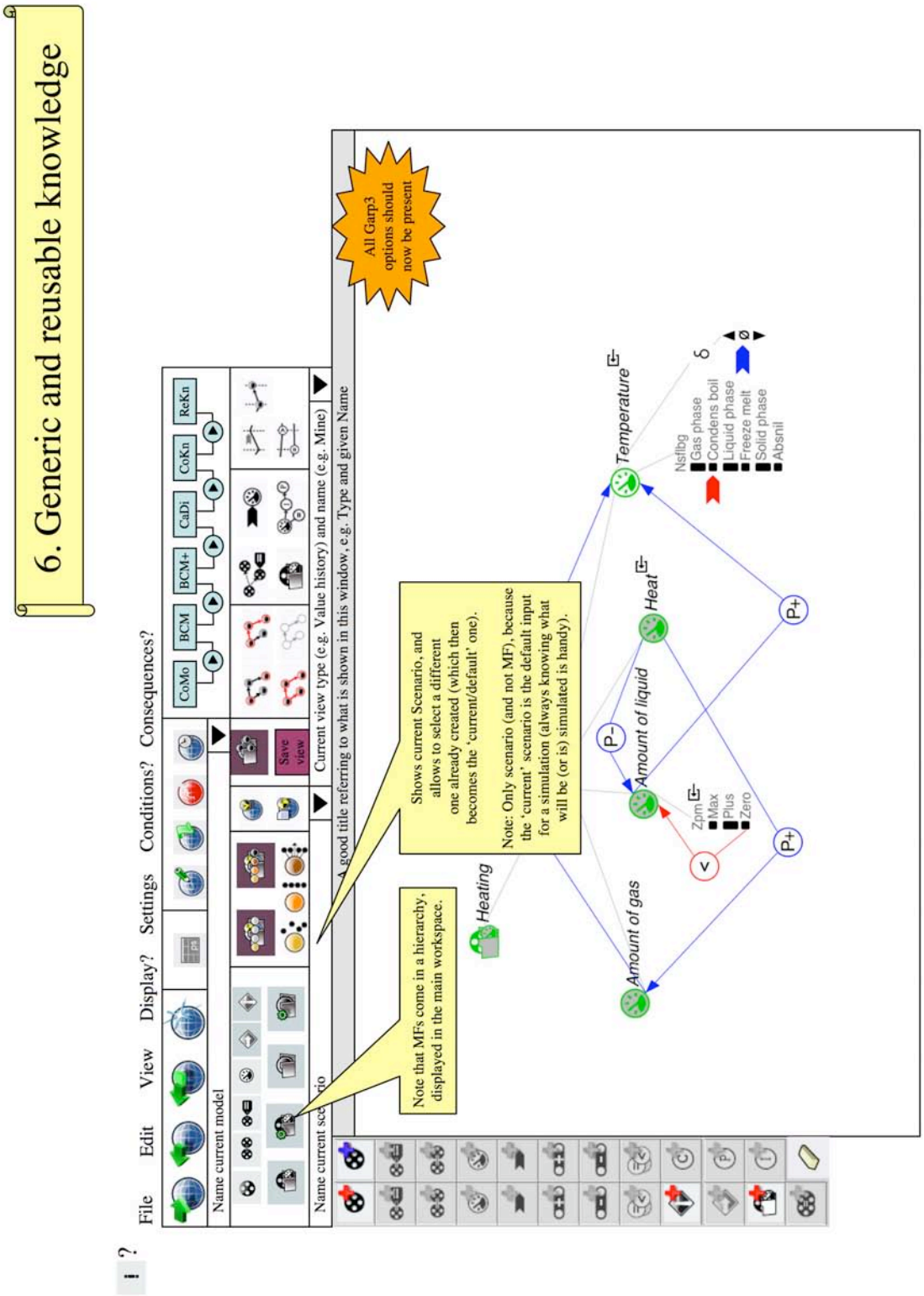

Figure A.6: Physical design Use-level 6 - Generic and reusable knowledge.

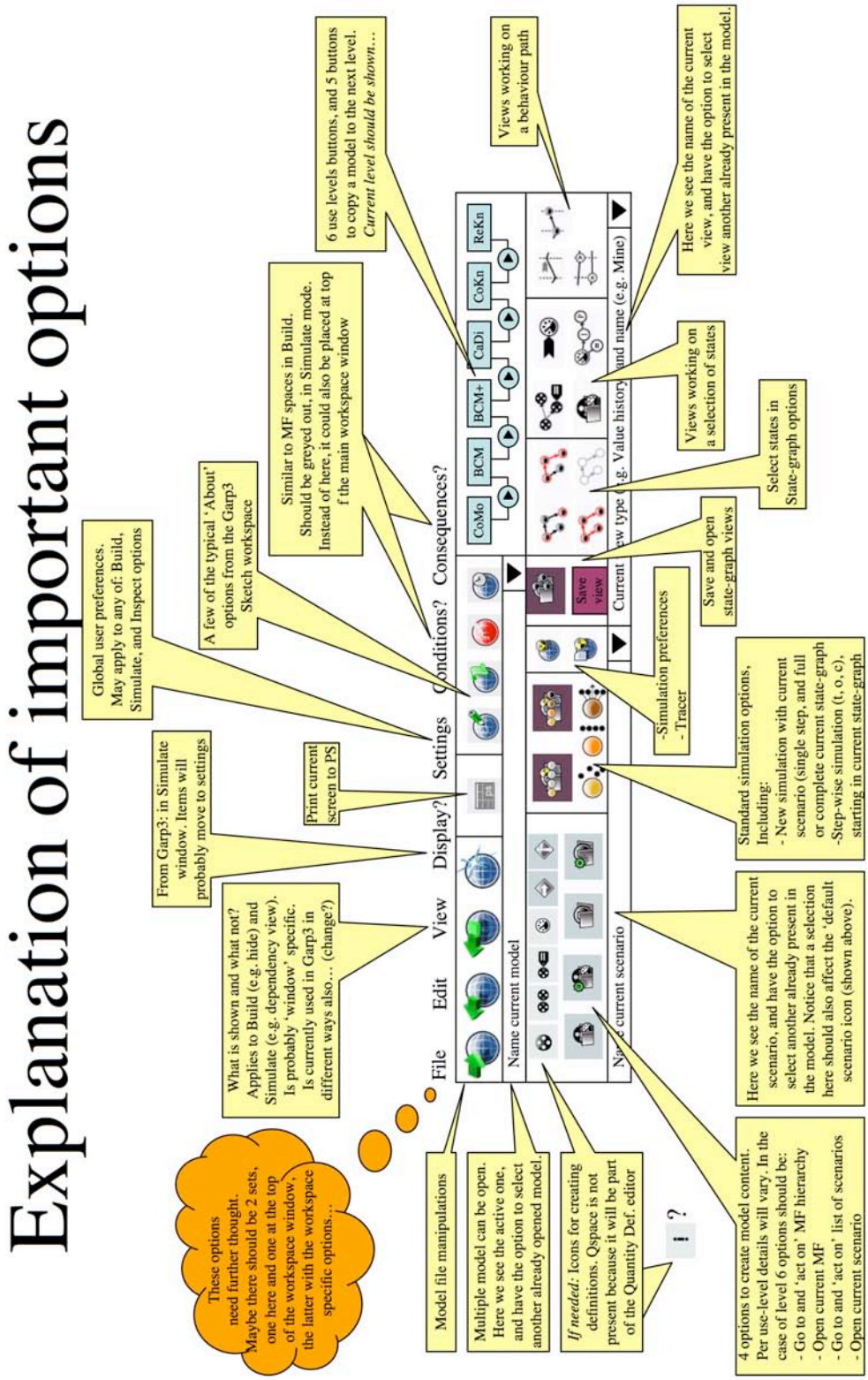

Figure A.7: Physical design - Explanation of all important options.

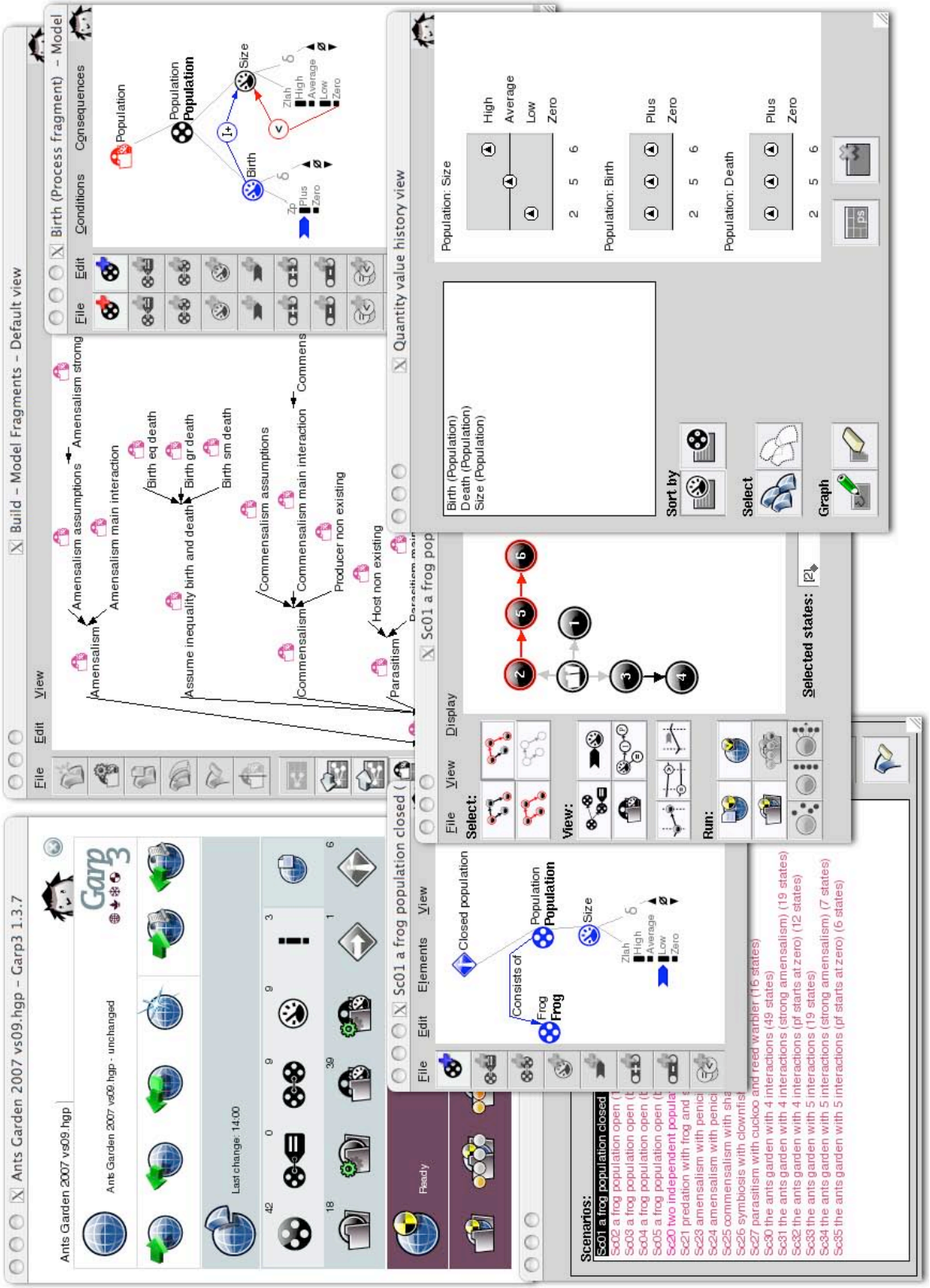

Figure A.8: Impression of the Garp3 workbench showing a simple model. The following screens are partially shown: top (from left to right): main screen, model fragment selector/overview, model fragment editor, and bottom (from left to right): scenario selector/overview, scenario editor, simulate main window (state-graph), value history.

Appendix B – Handling Garp3 add & definitions dialogues

Use-level 1

Only use simple selection and creation list for nodes (entities) and links (configurations). Garp3 detail: all entities are default placed directly under the top node entity (which itself is not shown).

Use-level 2

Entities, configurations, and quantities should come via simple selection and creation lists, similar to level 1. Attribute requires the options to *define* the different values for the attribute.

Notice: we could just as well leave attributes out of this level. This also requires that in the top-level screen the attribute icon will be removed. Let's make this change, because ultimately it will be easier for learner (having the attribute doesn't add much from a learning point of view)

Use-level 3

Essentially similar to level 2 (the attribute is removed here also). Extra at this level is the quantity space. There will be a set of default options, and learners can decide to create new ones. Selecting a default should be an option (adding something to the canvas) when a quantity is selected (functionally compares to: adding a quantity when an entity is selected). Some default settings may be handy, that is, if the learner wants to add a quantity space to volume1 and volume2 already has zmp, than the zmp would be the default selection when starting to add a quantity space to volume1 (but learners may decide to use a different quantity space), etc. Creating your own quantity spaces should be an option that you can select in the add-quantity space 'dialogue'.

Notice: deleting a quantity space from a quantity should be an option!

Use-level 4

Essentially similar to level 2 and 3, for entities, configurations, and quantities (attribute is removed). Using quantity spaces should also be the same as for level 3. Assumptions and agents should be handled similar to entities, with simple add/create lists. All created ingredients of these types are placed directly under the top node, and the top node is not shown as a selection option. As a reference, at this level it should be possible to use all ingredients available in Garp3, but now within a single screen, while all typing is potentially hidden, and both scenarios and model fragments are not used explicitly by the learner.

Use-level 5

Similar to level 2 and 3 and 4, for entities, configurations, and quantities. Attributes should now be added, and quantity spaces need to be assigned to a quantity *before* the latter can be added to the canvas.

Use-level 6

Creating and using definitions works as in Garp3, but we would like to optimise, simplify the use of interactive windows as much as possible.

How/where to place the definition-screens and the add-screens?

It concerns the following types:

- Entity (and agent and assumption)
- Attribute
- Configuration
- Quantity (and quantity space)

Option 1

1. Selecting an ingredient in the canvas leads to highlighting possible add options (as in Garp3)
2. Selecting one of the highlighting options leads to an interactive dialogue:
    a. Make a selection (all other Garp3 options are blocked) and press add, the latter closes this window (learner is back to situation 1)
    b. (If available, depends on use-level) press 'edit definitions option', this leads to current 'add' dialogue to 'change into' definitions dialogue
        i. Learner creates definitions (all other Garp3 options are blocked) and closes dialogue when done, the latter leads to situation 2 (the add dialogue reappears).

Notice: option 1 is Garp3 style but the appearance/timing of dialogues is slightly different. The 'dialogues' windows are places on top of the workspace (as now done on Garp3, and will remain on top, until they are closed).

Option 2 (more complex to make, interface wise speaking)

The icons on the menu bar (LHS) are 'place holders' for the add dialogues. Instead of these icons getting highlighted (depending on the selection in the workspace). They open up into the add-dialogues depending on the selection in the workspace.

When open (available) we have situation 2 (as mentioned above), but the dialogue is on the LHS of the workspace connected to the canvas somehow. You can now do option 2a, or make a different selection in the workspace (option 1). Different is that learners do not have to close the window in order to do step 1 (as opposed to: in the case of option 1 finalising the add was essential).

You can also do option 2b (this window stays on top and has to be closed before anything else can be done (as above for b2). We are not sure yet where the window would be placed (to do the definitions), but probably it could be a stand-alone screen as in Garp3, that appears on top of the canvas. Alternatively, it could be a 'morph' of the add dialogue (replacing this add dialogue, or sitting partially on top of it), but this may not be clear enough. So better have it seriously on top in the middle of the canvas. Another options would be to place these windows fully in the workspace (similar to how model fragments and scenarios will be handled), but that will probably be even more confusing.

Notice: Definition icons top menu bar open definition dialogues directly. Closing should then lead back to situation 1 (option 1).

Concluding remark

Option 1 seems best and most easy to accomplish.

e-mail: Info@DynaLearn.eu
website: www.DynaLearn.eu