Supporting the Construction of Qualitative Knowledge models
Bessa Machado, V.

Citation for published version (APA):
Bessa Machado, V. (2004). Supporting the Construction of Qualitative Knowledge models

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Design and Implementation of MOBUM

5.1 Introduction

This chapter describes the design and implementation of MOBUM, an environment that supports users in constructing models of real world systems, allowing users to play with them and test them by means of simulations. MOBUM was implemented using JAVA [81]. The system is based on the principles presented in Chapter 2 taking into account the lessons learned from the experimental evaluation of HOMER (Chapter 3). Just as HOMER, MOBUM consists of a set of builders providing assistance in the various phases of the model building process. Both tools differ in many respects though. The first difference lies is the number of context-specific builders supporting the formal definition of the model, which has been reduced in MOBUM. On the other hand new builders were introduced, which do not add contents to the model but are intended to assist the modellers in better understanding the system they are building a model of. Also, the interaction with the builders occurs in a different fashion. The interaction of the user with the tools for creating a new model ingredient, for instance, will always require user interaction with a dialogue which is linked to the tool in order to complete the task. Finally, in MOBUM all builders are equipped with support capabilities, by means of help agents which assist the user in each of the model building steps. These and other differences between both tools will be further discussed in this chapter.

The architecture of the model building environment is described in Section 5.2. The design of the User Interface is then presented in Section 5.3. This module receives the main focus of attention in this chapter. The components are explained with reference to the design guidelines as presented in Chapter 2. This includes the description of builders, tools, and the Help System. Section 5.4 describes the technical aspects involved in the implementation of MOBUM. Section 5.5 concludes this chapter.
5.2 System Architecture

A client-server organisation has been chosen where the client supports the modelling tasks and the server provides the simulation engine. Figure 5.1 depicts the architecture of the overall model building environment, which consists of a Graphical User Interface (GUI), a Model Representation module (GKOM) and a Model Execution module (simulator GARP). The communication between the Model Representation and Model Execution modules is realised through XML documents.

**GUI** The Graphical User Interface comprises three subsystems, *Mobum*\(^1\), *Toolbox*, and *GarpApplet*. The Mobum subsystem consists of a set of builders that support the construction and manipulation of model ingredients in different steps. Some builders represent models as *graphs* and therefore their model ingredients are represented by *nodes* and *arcs*. Other builders, which support the definition of individual model ingredients, organise the model ingredients as items in a table. The Toolbox system provides means for expressing ideas by using drawings and causal modelling tools. The GarpApplet enables the simulation of a scenario and the visualisation of the simulation results. Additionally, GarpApplet provides means for inspecting the simulation, for instance, by verifying which Model Fragments were triggered during the simulation.

All builders and the toolbox are equipped with a help system providing dedicated support. A detailed description of the GUI is given in Section 5.3.

---

\(^1\)The subsystem Mobum refers to one part of the whole system (the main application and builders), which is also called MOBUM.
GKOM  The Model Representation encompasses primitives for storing models and simulation results. The GKOM\textsuperscript{2} library\cite{111} constitutes the system model representation module, see Section 5.2.2.

GARP  Model Execution refers to the simulation engine, GARP, which can run simulations using the models constructed by means of the GUI and which are stored in GKOM. The client holds the GUI and the GKOM library, which are both written in Java. The simulator (GARP), on the other hand, which is stored at the server side, is written in Prolog. Java has been chosen for the implementation of the application for a number of reasons:

- it is platform-independent,
- it allows easy implementation of an effective GUI,
- it supports a number of network communication mechanisms, particularly through the Internet.

The most important benefit that model builders may gain from the client-server design is the possibility of exchanging models. Different users, at distinct geographic locations, for instance, can store their models on the server and other users can (re)use these models in their work. Additionally, we have the ability to easily maintain and control the simulation engine. In the following sections a more detailed description of the components of the system architecture is given.

5.2.1 Model Execution and Communication

The model execution module refers to the qualitative simulation tool. After creating a model of a system, the modeller will probably wish to test it by simulation. For that purpose, the simulator will receive the model as an input and will produce a behaviour graph (consisting of qualitatively distinct states) as an output. An analysis of the simulation tool, GARP, was already given in Chapter 2.

The communication between the client and the server comprises two aspects: exchanging model and simulation data and controlling the way in which the simulator runs the model. The exchange of model and simulation data is realised using an Extensible Markup Language (XML) document. XML is widely used for describing documents as well as other types of data. An interesting feature of XML documents is that they do not include formatting instructions. In this way, the same data can easily be displayed in various manners. Additionally, XML parsers exist for most of the major programming languages. For these reasons XML is the natural choice for arranging the communication between the model building environment and the simulator.

In the following paragraphs, we first describe the communication aspects that concern the representation of the model. Thereafter, a description of the available control over the simulator via the communication module is given.

\textsuperscript{2}Garp Knowledge Object Model
Communicating via XML and DTD  XML adopts a tree-representation of the underlying data and is considered self-describing, i.e., XML data does not require any predefined schemes. However, it is possible to impose structure onto XML documents by using the Document Type Definition (DTD) or other schematic languages (e.g. XML Schema). A DTD defines, for example, which tags are permissible in an XML document, the order in which such tags must appear and how the tags are nested to form a hierarchical structure.

A relation between two elements can be specified in two ways: by nesting or by referring to the identifier of an element, which must be unique for each element. As an illustration, consider the following situation: A quantity Level belongs to the entity Liquid and has associated with it the quantity space zp (zero and plus). The following example illustrates how a DTD can be specified to represent generic quantities, defined as QuantityType, in the model as well as a valid XML document for the DTD.

```
1. <!ELEMENT QuantityTypes (QuantityType*)>
   <!ELEMENT QuantityTypes EMPTY>
   <!ATTLIST QuantityType quantityType ID #REQUIRED
   entityTypess IDREFS #REQUIRED
   quantitySpace IDREF #REQUIRED>

2. <QuantityTypes>
   <QuantityType quantityType = "level" quantitySpace = "zp" entityType = "liquid"/>
</QuantityTypes>
```

Figure 5.2
1. Sample DTD, 2. XML Source.

The names of the elements and attributes and their order in the hierarchy (among other things) constitute the XML markup language used by the document. In the example shown above, the language contains the element QuantityType, which defines generic quantities and contains three attributes:

1. quantityType gives the name of the quantity ("level");

2. entityType refers to the list of entityType that the quantity may belong to. This attribute refers to the idref data type (EntityType).

3. quantitySpace contains a reference to the quantity’s quantity space ("zp").

In short, the DTD describes all the elements needed to represent a model that can be used by the simulator for predicting states of behaviour.

Controlling the GARP server  Next to the exchange of model information, the communication with the server allows a client to specify the actions that the server is supposed to perform. The following options are available in this respect.

- Load a model. The server receives a model and waits for further instructions.
5.2. SYSTEM ARCHITECTURE

- **Select a scenario and run it.** If a scenario is selected from the list of scenarios, the server can be requested to run that scenario. In fact, running a scenario may be done in two modes: either a request to run the first interpretation of that scenario, which may lead to initial successor states; or run the scenario while requesting full simulation, which will generate the complete behavioural graph.

- **Terminate, Order and Close.** If a full simulation has not been chosen, GARP supports choices in regard to which transitions are to be further explored. For this purpose the following controls exist and should be called sequentially (for selected states or for all states): terminate, order and close. **Terminate**, based on a state content, will determine a set of possible terminations, including how quantity values and/or inequalities may change as a result of derivatives being non-zero. **Order** will order and remove terminations using a set of rules specifying ordering information. **Close** determines the transition to a successor state, if available.

- **Run full simulation.** Full simulation can also be requested for specific states or for all states. In this way, the sequential control steps above, terminate, order and close, are all performed. This will generate a complete state transition graph.

- **Request Simulation Information.** After running a scenario, the server can provide lists of information of the simulated model. The following information can be requested: complete model contents or partial model contents. In the latter case, information regarding either specific states or all states and how transitions occurred from one state to another can be obtained.

5.2.2 Model Representation: GKOM/Object Model

At the client side, a model is represented by the GKOM library [111]. In the following the main packages composing the GKOM library are described in detail.

**gkom.parsers** This package implements the import-export module to the GKOM Library. It contains the classes\(^3\) responsible for taking a GKOM knowledge model, which was created using the classes of the gkom.model package, and transforming it into an XML document representing the main categories of knowledge that are input to GARP. Also it contains the classes that produce the reverse action, i.e. it contains classes that handle XML documents representing GARP knowledge models and has methods for transforming them into GKOM knowledge models.

In order to run a simulation, it is required to provide a set of entities organised as an is-a hierarchy, the model fragments, the quantity spaces, and at least one scenario which will be the starting point for the simulation. The XML document produced contains all these and will be sent to the server.

---

\(^3\)This notion of classes refers to classes as used in JAVA and should not be confused with the notion of class as discussed in Chapter 2. The Type level represents objects that correspond to the model primitives (classes) in GARP, such as entities and quantities.
**gkom.model**  Contains the primitives for creating and storing a model and simulations ran with that model (entities, relations, quantities, dependencies, model fragments, scenarios, etc). Figure 5.3 shows the top level classes of this package. The `KnowledgeObject` is an abstract class and the top class of the hierarchy. It has two attributes, name and description, which can be used by its subclasses. In the following paragraphs the top level classes of GKOM are described.

**Three Levels of Knowledge**  In GKOM, three levels of knowledge have been defined to represent a model: the `Type` level, the `Occurrence` level and the `Instance` level. The Type Level represents primitives as defined in GARP. The Occurrence Level represents any primitive that has been specified within model fragments. The Instance Level represents the primitives in the scenario and the contents of the behaviour graph as produced by the simulator.

As a result, GKOM uses three different classes to represent each unique primitive of the simulation model: A `Type` class, an `Occurrence` class and an `Instance` class. As shown in Figure 5.3 all Occurrences and Instances have a specific type. The simulator uses the Instances specified in a scenario to create a full model. This result is then used by the parser to create the Instances in the GKOM model. An Instance holds information specific of the different states it may be in and can (of course) not be modified by the modeller. The following example may help in clarifying the utility of this kind of representation. The generic quantities (e.g. temperature) of the model are defined in the Quantity Builder when using MOBUM. What is in fact created at this stage are `QuantityTypes` objects. When such a quantity is assigned to a model fragment, an `Occurrence` of that quantity, a `QuantityOccurrence`, is created. Finally, a `QuantityInstance` represents the existence of the quantity in a scenario and (after simulation) the state in which that quantity exists. A quantity may of course have different values in different states.

The `Model` class represents a complete GKOM model. Models are the starting point of every model building activity. The main functionality of the class is to hold the generic knowledge of a model as well as the results of simulations based on that knowledge. One of the features of the Model Class is that it controls the creation and destruction of Types. By having the Model manage the creation and destruction process, the consistency of
the whole model is guaranteed. When a Type is created, it must always be added to a Model. The Model then will check the validity of the newly created object by verifying the uniqueness of the object. Another responsibility of the Model class is the deletion of model ingredients from the model. It does not allow, for instance, the modeller to remove a Type if anywhere in the model there is an occurrence of that type. If the modeller really intended to remove that Type, he or she must first remove the Occurrence of the type be

![Diagram](image)

**Figure 5.4**
Model class diagram.

Figure 5.4 shows the main hierarchical organisation of the class *Model*. Figure 5.5 shows the classes ScenarioType and ModelFragmentType in more detail. Some extra classes have been created, in order to effectively represent the simulation model. Class Block, for example, has been implemented to hold the *Occurrences* created in the context of model fragments and scenarios. In fact, the class Block and its subclasses ConditionsBlock and ConsequencesBlock were introduced to facilitate efficient implementation of the domain without compromising the independence of the domain from a specific user interface. All classes are part of package gkom.model and are, except for the Block class, derived from class KnowledgeObject.

The *Simulation* class represents the simulation results produced by the simulator based on a specific Scenario. Simulations are created by GKM during the process of parsing the output of the simulator. A *Model* can hold one simulation for each created *ScenarioType*. A *Simulation* holds *State* and *Transition* objects resulting from the simulator. These will be constructed by the *Simulation* which will also check their uniqueness and thus guarantee that no duplicates are created. In this way, an instance that appears in different states is represented just as only one instance with references to the states in which it exists.

The *State* class represents a state of behaviour which has resulted from a simulation. As shown in Figure 5.3, all states produced have a reference to the Simulation they belong to. A State is identified by its number and has the information of all *Instances* present in that State. A State is the result of a simulated model and State object is created when parsing the results.

The *Transition* class represents the knowledge about how one state moves to another
CHAPTER 5. DESIGN AND IMPLEMENTATION OF MOBUM

Figure 5.5
ScenarioType and ModelFragmentType class diagram.

state. A Transition occurs based on rules. Transitions are also created by the parser based on the simulation results and are also owned by a Simulation object.

In short, a model builder will build a model which consists of Types and Occurrences. An Occurrence has one and only one Type but a Type may have \( n \) Occurrences. Instances, Simulations, States, Transitions are created by the simulator and added to GKOM by the parser (of the simulation results).

gkom.server Contains classes that handle the communication with the remote simulation module. The classes in this package make up the communication module.

5.3 User Interface

MOBUM is intended to support users in constructing system models, allowing them to play with these models and to test them by simulation. The access to the various functionalities of MOBUM is made through the buttons on the primary toolbar (see Figure 5.6). In the following sections, the individual tasks and actions that can be performed are described.

Figure 5.6
MOBUM main window.
As was the case in HOMER, the user interface of MOBUM has been designed following the general idea of not allowing the user to construct a syntactically erroneous model. Therefore, only a limited set of actions is possible at each stage of the model construction. However, the user is free in determining the order in which every ingredient of the model is created. The visual representation of model ingredients within the builders follows the guidelines presented in Chapter 2. For example, Quantities in the Quantity Builder are organised in a list, because no relation exist between quantities, while Entities are represented as nodes in a graph and the is-a relations between the entities represented as arcs between those nodes.

The model building environment uses two major notions which were described in Chapter 2: the builders and the tools, and additionally, the environment incorporates model inspection tools.

Builders are interactive windows that support the user in building specific model ingredients. In the current version of MOBUM there are five builders that support the creation of specific model parts, namely: Model Fragment Builder, Quantity Builder, Quantity Space Builder, Structure Builder and Scenario Builder. Additionally there are two builders that do not add directly content to the model but support the modeller in exercising his or her conception about the system being modelled: The SketchPad and the Causal Builder. Multiple builders can be simultaneously open while constructing a model. Tools are the means of interaction with a specific builder. The set of available tools for the builders are displayed at the secondary toolbar on the left hand side (LHS) of each builder and can also be accessed from the menu of the builder.

Model Inspection Tools are the available tools for the modeller to run a simulation and also to inspect the results of the simulation. After running a simulation, the modeller will get a behaviour graph and can investigate, for instance, how the quantities behave in the different states produced, which model fragments applied, the content of a specific state, how the transition occurred from one state to another.

The remainder of this section details the design of each builder and inspection tools in MOBUM. We start with the five builders which have common characteristics with the builders in HOMER. Subsequently, the description of the new functionality is given. This consists of extra tools designed with the purpose of making the task of building qualitative models more insightful, the SketchPad and Causal Builder. Finally, the model inspection tools for the analysis of simulation results is described.

5.3.1 Structure Builder

In the Structure Builder the user defines the entities, structural relations, and attributes. The entities should be hierarchically organised in a top-down manner, which means that the definition of the most general domain concepts should be given at the top and subsequent specialisations of the concepts should follow below. After having been defined in this builder, model ingredients can be reused to construct model fragments and scenarios. We decided to create structural relations and attributes in the same workspace. Conversely, in HOMER dedicated builders for defining attributes and structural relations were used. During the experiment with HOMER, it was noticed that the subjects did not
grasp the utility of defining them separately. Furthermore, all these concepts refer to the *structure* of the system being modelled and as such it seems a natural choice to put them together.

![Figure 5.7](image)

**Figure 5.7**

Structure Builder.

Figure 5.7 shows a possible *structure* for the U-Tube system. It consists of entities: *substance*, *path*, *liquid*, *fluid path*, and *container*. The entity 'fluid path' has the attribute *status*, which may assume values such as *aligned* and *not aligned*. The structural relation *contains* relates entities 'container' and 'liquid'. The figure also illustrates the way in which the builder and the available tools interact. By selecting two entities, *fluid path* and a *container*, only one action is possible: defining a structural relation. By using this context-sensitive approach, only syntactically correct choices are made available. Notice that only the last option in the toolbar of the builder has been selected (highlighted). Therefore, the interaction dialogue associated with the tool is displayed with the selected entities already in place. Finally, the user must type the name of the structural relation that is being defined.

### 5.3.2 Quantity Builder

In this builder the quantity types are defined. Each quantity has a quantity space, which is defined in the Quantity Space Builder. The quantities defined in this builder can be used to construct model fragments and scenarios.
5.3. USER INTERFACE

Comparable to HOMER, the user can initiate the creation of a quantity space when creating a new quantity, see Figure 5.8. Contrary to HOMER, if the user does so, the created quantity space will be, by default, assigned to the quantity being created. Therefore, no further choices need to be made. Like in the Quantity Space Builder, the results are displayed in lists.

![Figure 5.8](image)
Quantity Builder.

5.3.3 Quantity Space Builder

In the Quantity Space Builder the user defines the set of possible values for the quantities. This builder can be used stand-alone or activated from within the Quantity Builder. Quantity spaces may be reused and therefore can be associated with one or more quantities. A quantity space is assigned to a quantity in the Quantity Builder. One feature of this Quantity Space Builder that differs significantly from the Quantity Space Builder of HOMER, is the way of creating and editing a quantity space. In HOMER a quantity space is created and edited inside the workspace that holds the list of existing quantity spaces. In MOBUM, the Quantity Space Builder holds the list of existing quantity spaces. A quantity space is constructed by means of a creation tool designed for that specific purpose. There are several reasons behind this design choice. Firstly, the builder is kept clean and more insightful. Secondly, having the creation in a dedicated window may help users in focusing their attention on the task being performed at that time. Thirdly, we wanted to keep a high coherence. If the paradigm is to always have a tool associated with an interaction dialogue, this should be true for the whole system.

Figure 5.9 shows the Quantity Space Builder together with the dialogue associated with creating/editing a quantity space. Similar to HOMER, the builder organises the
created quantity spaces as a list and by selecting one of the items in this list its preview is displayed in the column besides it. In this way, the user can visualise the defined values and their types.

![Image of Quantity Space Builder](image)

**Figure 5.9**  
Quantity Space Builder.

### 5.3.4 Model Fragment Builder

In the Model Fragment Builder the user will model the knowledge about the behaviour of entities that have been specified in the Structure Builder. This includes the specification of features of entities, such as quantities and the values they assume. Constructing a model fragment means reusing the model ingredients previously defined in the Structure, Quantity Space and Quantity Builder.

Some differences exist between the model fragment builder of MOBUM and its counterpart in HOMER. Firstly, the way of navigating through the builders. In HOMER, when adding a model ingredient to a model fragment or scenario, the user may access the builder associated with the model ingredient from the "Add window" and still keep the "Add Window" open. In MOBUM, we did not implement such a feature. Furthermore the dialogues associated with available tools within the builders are modal. This means that, the user must finish or cancel the actual task before being able to initialise a new one. In fact, one outcome of the experiment with HOMER was that the subjects frequently got lost when doing this, initiating one task before finishing the previous one. Another difference is that we do not have "Conditions" and "Consequences" as separate menus. In MOBUM a check-box is provided within each 'Add dialogue' to switch the location of the model ingredient from *Condition* to *Consequence*. When a model ingredient can only
be specified as a consequence, the option is unavailable. Also, when adding a quantity, the modeller may at this point specify the actual value and derivative of that quantity (see 'Add Quantity' dialogue in Figure 5.10. In HOMER, these tasks are independent and placed within the 'Conditions' and 'Consequences' menus. When using HOMER, the subjects selected values from the quantity space. Although the value was highlighted it did not mean that it was selected. In the beginning, this was the cause of some confusion among the subjects.

Figure 5.10
Model Fragment Builder and 'Adding Quantity' dialogue.

5.3.5 Scenario Builder

This builder is used to create situations that will be given to the simulator for behaviour prediction. The partial models created using the Model Fragment builder represent generic domain knowledge while a scenario contains instantiated knowledge describing a particular state of the simulation model. The way of representing generic knowledge, as it appears in a model fragment is similar to the one used to represent a particular state in a scenario. As a consequence, the tools that apply to the Scenario Builder and the Model Fragment Builder are similar.
5.3.6 Model Browser

The model browser, Figure 5.11, is the first component that does not have a counterpart in HOMER. Its functionality consists basically of displaying a global overview of all the model ingredients that have been defined in the current model. It compactly shows, for example, quantities, relations, the hierarchical organisation of entities and model fragments. By double clicking on a model ingredient the associated builder will be opened. Similar to other system browsers, the Model Browser does also perform standard actions such as deleting or editing the name of objects.

![Model Browser](image)

Figure 5.11
Model Browser.

5.3.7 Toolbox package

The Toolbox package encompasses tools that support tasks not directly involved in the formalisation of the model. First, it implements what we have coined *intermediate modelling support*. Often when building a model, the persons building it define intermediate models before they write down the final model. For instance, they make drawings in order to visualize/understand the system they are working on as a whole. Second, during the experiment with HOMER, it was noticed that all the subjects were frequently trying to follow the behaviour of the system by mentally constructing the causal model of the
system. We interpreted this as an indication that a tool was needed to provide the means for the user to construct a dependency graph. It should be noticed that this is even more important when the system being modelled includes many quantities and several dependencies between them. This tool would support the user in investigating if and how the quantities interact through the causal dependencies.

**SWAN SketchPad**

This is a brainstorming tool for expressing ideas by making a drawing of the situation to be modelled, see Figure 5.12. Although the tool does not add any actual content to the model, if the modeller classifies the individual parts of the drawing using the classification tools, this information will be known to the help agents and they may help the modeller at the time of formalising the model. Besides creating a drawing and specifying the basic types and names of the objects within the model, the SWAN Sketchpad allows the creation of scenes which represent the envisioned states that the user wishes to distinguish.

![SWAN SketchPad](image)

**Figure 5.12**

SWAN SketchPad.

In fact, the movie scene metaphor is employed in Sketchpad. Each expected state in the simulation can be represented as a scene. The modeller can make a copy of the current drawing and modify aspects in order to express expected behaviour. This visualisation makes the conception the modeller has of how the simulation should behave more explicit. The SWAN Sketchpad always has one original drawing, called the initial drawing. This is the drawing created during the sketching process. Additional scenes may be added to the SWAN Sketchpad and the filmstrip underneath is augmented whenever this occurs. It is important to distinguish between the two types of drawings; the original drawing describes the static features of a system, the scene drawings describe the changes. As an example, consider Figures 5.12 and 5.13. Figure 5.12 shows an initial drawing of the U-tube system. Figure 5.13 shows two scenes created and already manipulated. The figures
tell us that if we start with the situation where the LHS container has more liquid than the RHS container, in the next state the system may reach an equilibrium and, therefore, the two containers will have equal levels of liquid.

![Figure 5.13](image-url)

Two scenes of the U-Tube.

In order to interact with the SWAN SketchPad, the user can use a set of tools, which are placed on the LHS toolbar of the workspace. Some of the tools are standard and common to other drawing tools. Looking at the toolbar, from top to down, the following tools exist:

- **New Scene**: This tool creates a new scene of the actual drawing.
- **Remove Scene**: removes the current scene from the drawing.
- **Standard Drawing Tools**: Selection Tool, Text Tool, Rectangle Tool, Round Rectangle Tool, Oval Tool, Line Tool, Scribble Tool

*Special purpose Tools*: The following tools are used to classify a drawing. The classification is made by setting a (model) type to the graphical object created.

- **Set Type to Entity**: This tool classifies a selected text or shape as an entity, for instance, in the U-Tube drawing container, liquid, path would be classified as entities.
- **Set Type to Process**: Classify a drawing part as a process. For instance, the arrow indicating the flow of liquid from one container to another would be classified as a process.
- **Set Type to Quantity**: set the type as a quantity. Looking at the U-Tube drawing Level and Pressure would be set as quantities.
- **Remove Type**: remove a set type from a drawing or text.
- **Group Selection**: Group a selection of shapes and/or texts.
5.3. USER INTERFACE

- **Ungroup Selection:** Ungroup the current selection of shapes and/or texts.

Additionally, the user can manipulate the attributes of drawing elements. Text can have different fonts, sizes and colour. Shapes can be filled and also have different colours. The line thickness, colour and arrow specifications can also be changed using the shape attributes.

**Causal Model Builder**

This builder is a brainstorming tool to help the user express ideas by drawing a causal model. Similar to the 'SWAN SketchPad', it does not add any actual contents to the model. Figure 5.14 illustrates a causal model drawing for the U-Tube system. The goal here is that by including all quantities and the dependencies between them, the modeller may acquire an overall picture of the behaviour of the system. Therefore, the work of formalising the model can be performed more easily.

![Figure 5.14](image)

Causal Model Builder.

### 5.3.8 GarpApplet: Inspecting Simulation Results

With GarpApplet it is possible to run a simulation. As already explained, the simulator is installed on the server. Therefore, when the user chooses to start a simulation, by selecting the GarpApplet on the main window, the user will get a dialogue box requesting him or her to select the server where the simulator is located, see Figure 5.15. One possible configuration is already given by default. The model is subsequently sent to the server. The server will prompt the existing scenarios of the actual model, also shown in Figure 5.15. After selecting a scenario, the user has a series of controls over the simulation. In Figure 5.16, looking at the toolbar, located at the top of the simulation window, the
GROUPS  OF  BUTTONS  SELECT  ALLOWS  THE  USER  TO  SELECT  ALL  THE  STATES  OR  ONLY  SPECIFIC  STATES.  THE  FOLLOWING  GROUP  OF  BUTTONS, ACTION, CONSISTS  OF  ACTIONS  THAT  CONTROL  THE  SIMULATION.  THE  FIRST  THREE  BUTTONS  OF  THIS  GROUP  COMPRISE  THE  FOLLOWING  CONTROLS:  TERMINATE, ORDER  AND  CLOSE.  THE  FOLLOWING  BUTTON  OF  THIS  GROUP  EXECUTES  THESE  THREE  ACTIONS  AT  ONCE.  THE  LAST  BUTTON  ON  THE  GROUP  RUNS  A  FULL  SIMULATION.

FIGURE 5.16
U-TUBE BEHAVIOUR GRAPH.

FIGURE 5.16 SHOWS  THE  RESULTING  BEHAVIOURAL  GRAPH  FOR  A U-TUBE SCENARIO  WHEN 'FULL
Simulation' has been requested. It consists of two states (the numbered circles). Figure 5.17 shows the table version of the graph and also includes the quantities in the system and how their values changed over time. At this point, the user can inspect the simulation results. When selecting a state, all the model ingredients within that state are visualised. It is also possible to filter information. For instance, it can be chosen to visualise only the quantities in the state, certain dependencies between the quantities etc. In Figure 5.18, State 2 has been selected and its information is displayed.

In the inspection window we can choose to visualise different model ingredients within the state. For instance, on the top of the window many configurations are available. After making a selection concerning what is involved (relations, dependencies, entities, and quantity spaces), the following options exist: *mouse*, when the mouse is over the model ingredient it will be visible otherwise it will be hidden, *all*, all existing model ingredients will be visible, *none*, none of the model ingredients will be visible. Also, we can verify which model fragments were used during the simulation. Why one state changed
Figures 5.18
State information.

to another state can also be investigated. Figure 5.19 shows the rules that were applicable for taking state one to state two.

Figure 5.19
Rules: Transition from 1 to 2.

During the whole process of simulating a scenario, a connection with the server will be kept alive in order to execute each of the above-presented actions (interactions with the simulator).
5.4 User interface implementation

5.4.1 Top-Level

Every builder contains a series of graphs and other simple graphical components, such as combo-boxes and labels. A graph is a collection of vertices connected by arcs. These are placed and manipulated by the user using a mouse-controlled tool. Every tool has associated with it an interaction dialogue box. The toolbar of each builder (and the menu) shows the set of the tools that may be used.

5.4.2 Builders

Figure 5.20 depicts the set of currently available builders in MOBUM. Except for the Quantity and Quantity Space Builders, which organise their model ingredients in a tabular way, all other builders and the Toolbox are based on the graphical framework called JHotDraw [49], a Java based drawing package. Every builder inherits from abstract class Builder and will manage a collection of graphs and other dialogue components. The specific layout and number of these components will be defined by the concrete class constructor. Each builder has its own menu and toolbar, both of which hold the main functionalities of the builder.

Class Builder inherits from class AvatarFrame, which provides access to the help system and will be discussed later.

![Class Diagram](image)

Figure 5.20
Implemented Builders in MOBUM.
5.4.3 Graphs

Graphs are graphical representations of domain concepts. Figure 5.21 shows the currently implemented graphs. They all inherit from class Graph and represent specific domain objects (e.g. GScenario represents class Block in the domain layer, see Figure 5.5). Every graph contains vertices, which may be connected by arcs.

![Graph Diagram](image)

**Figure 5.21**
Implemented Graphs in MOBUM.

In order to satisfy the strict de-coupling requirement between the user interface and the domain object model, while still being able to represent the actual state of the system, the observer pattern [48] has been employed as a concrete implementation of the Model-View-Presenter pattern. Figure 5.22 depicts the details of the observer mechanism. An observable object will have a list of observers interested in obtaining updates about its internal state. Every time, an observable object changes its internal state, it invokes the notifyObservers method. This method will go through the list of observers and will notify every single observer via an update message. This message will contain a reference to the observable object, an identifier of the type of event that occurred as well as information needed to completely describe the change of state.

In this way the observable object does not have any direct coupling to the observer. In GKOM, all domain classes inherit from Observable, while in MOBUM all graphical representations of domain classes implement the Observer interface.

An example will be given next. Consider the class diagram of Figure 5.23. Classes EntityType and Entity pertain to the GKOM Library domain, while GEntityType and GEntity are part of the user interface. Every object of class GEntityType will observe the internal state of the object of class EntityType to which it is associated. Whenever the name of the entityType changes via the setName method, all observers will be notified via their update method and will take appropriate actions. In this case the GEntity object will change its label to the new entityType’s name. A similar action is taken by an object of class Entity, which will propagate the name change event of the entity to which it is associated to its own observers, thus guaranteeing that the GEntity label of the object will be correctly updated.
5.4. USER INTERFACE IMPLEMENTATION

5.4.5.4 USER INTERFACE IMPLEMENTATION

5.4.5.4 USER INTERFACE IMPLEMENTATION

Figure 5.22
Observer Structure.

5.4.4 Vertices and Arcs

Every vertex may contain several graphic components and ports. A port represents the
interface between an arc and a vertex and may contain several snap point coordinates. Every
arc may also contain several graphic components, and an optional arrow tip at each end.
As with the Graph class in the previous section (and following the same convention for
giving names, i.e. G+"domain class counterpart"), implementations of concrete versions
of the vertex and arc classes representing specific domain classes exist in MOBUM. Fig-
ures 5.24 and 5.25 show the class library. These apply the model-view-presenter pattern,
previously presented, in order to represent the present internal state of the domain.

5.4.5 Support Module

A discussion of different approaches in supporting the model building task has been given
in Chapter 4. In this Section we will focus on the design and technical details of the help
system.

Overview

In MOBUM, six personalities, implemented as agents, are employed to create a definite
identity for each one of the types of static and dynamic support as specified in Chapter 4.
Each of the seven components in MOBUM, namely the five builders, the SWAN SketchPad and the Causal Model Builder, is equipped with its own specific version of the various agents. As shown in Figure 5.20, all builders inherit from class AvatarFrame. The class AvatarFrame is an abstract class that has abstract methods to trigger the help system of the model-building environment. Each builder implements its specific version of the methods thereby making the help system context-sensitive, meaning that help is provided based on the builder that is currently being used.

In the following, we describe how the various forms of static and dynamic user help are generated.
Static Help

Technically, the static help system is implemented by means of HTML files which are displayed inside a dialogue box. Each time one of the static agents is triggered by an event, the source of the event, i.e. the builder in which it occurred, is sent as one of the arguments of the event making it possible to invoke the builder's concrete implementation of an agent, which will deliver a tailored HTML file with adequate help for the situation at hand.

Figure 5.26 shows an example of a static help dialogue box which was displayed upon clicking on the How to agent in the Structure Builder.

![Static Help Example](image)

Dynamic Help

The dynamic agents provide advice to the user based on the current status of the model being built. In order to generate advice, the agents need a kind of intelligent reasoning engine. Furthermore, they are capable of performing an analysis spanning several different builders within the model-building environment and therefore will need access to an effective communication mechanism.

The communication and reasoning mechanisms are described in more detail below.

Communication In order to communicate knowledge there must be a common internal representation of communication. This common internal representation is provided by the Agent framework. It implements a formalisation of knowledge that facilitates communication. The framework provides a set of basic building blocks representing specific knowledge facets. All knowledge communicated is transformed into a common
form by means of these building blocks.

Building Blocks

The framework of the agent-based help system consists of knowledge objects, or features, each capturing a chunk of information that eventually may become an advice to the user. These features can be instantiated and placed in a hierarchical structure, or ontology. The position in the hierarchy is determined by the type of the knowledge object. Furthermore, these features can have attributes. These attributes are key-value pairs which can hold properties of the knowledge objects. Think of properties such as colour. Associations, or predicates, between knowledge objects are used to depict relations between the knowledge objects.

A predicate connects one or more knowledge objects with a named relation. One way to think of predicates is to see them as groups of objects that belong to each other. Predicates, features, types and attributes together form an ontology that facilitates the exchange of knowledge. Since one common ontology is used for exchanging knowledge, a common ground of communication between all participants is established. The entire set of primitives is described as:

Features The basic knowledge object is called a feature. This knowledge object always has a type. This type determines the position of the object in the ontology. A feature preferably has a name, however this is not obligatory.

Types As mentioned above, the type determines the hierarchical position of the feature object in the ontology and is unique. The isa hierarchy is specified at a global level, no specific local hierarchies exist. One root type element must be defined, it acts as the top-level object. All elements must have a name. All except the root element must have a parent type.

Attributes A feature can have a set of attributes. These are key-value pairs that can be set and read at any time.

Predicates Features can have associations. Predicates describes these associations. A predicate establishes a link to one or more features. These associations describe a kind of relationship. For instance, a group relationship can be expressed using a predicate between the grouping feature and the elements of the group.

Implementation

Figure 5.27 depicts the translation architecture in the modelling environment. The three helper classes are responsible for the transformation of their internal knowledge representation into the knowledge representation of the Agent API. These classes transform specific knowledge into features with types, attributes and predicates. Considerer the following example: a graphical shape classified as Object in the SWAN Sketchpad will be represented as a Feature of Type 'object' (any.swan.figure.object). A text label overlapping this shape would be represented as a feature of type 'text' (any.swan.figure.text).
5.4. USER INTERFACE IMPLEMENTATION

A predicate would represent the overlapping relationship. The predicate would be called 'overlapping' and would refer to both the 'text' feature and the 'object' feature.

**Reasoning** In order to provide intelligent support, certain reasoning capabilities must be included in the support system. This section discusses how advice, i.e. new knowledge, is inferred. So as to be capable of communicating with the various agents, the reasoning engine uses the Agent framework's common knowledge representation as described above.

**Rules**

In order to derive new knowledge, the knowledge governing the inference process needs to be described. Inference knowledge is formally described by means of rules. The inference engine uses the common knowledge representation of the framework, i.e. rules are given in the form of features, types, attributes and predicates.

Every rule consists of a condition and a consequence, which act respectively as the *if* and *then* parts of an equivalent *if-then* statement. When a condition is met its associated...
consequence becomes true.

Rules apply to a subset of features. This subset must be declared upon the definition of the rule. The names of the features and types must be provided. These features will be visible only within the scope of the rule, that is to say no global features can be defined.

*Conditions* consist of one or more *statements*. A statement can be given in the form of either an equation relating attributes of the features, or a predicate on one or more features. Statements are evaluated to be either true or false. An equation relating attributes is true whenever the equation holds. A predicate statement is true if the predicate exists in the knowledge base. If more than one statement exists in a condition, they must be connected using *connection symbols*. The available connection symbols are 'and' and 'or'. The 'and' symbol describes a *conjunction*, that is, in order for the 'and' to be true, both of the statements connected by the 'and' symbol must be true. The 'or' symbol describes a *disjunction* and is true if at least one of the statements it connects is true. Furthermore, a statement or a group of statements can be "booleanly" negated.

*Consequences* consist of one or more predicate statements and/or attribute assignments. Predicates will be added to the common knowledge base if the rule applies. Similarly, attributes will be assigned if the rule applies.

Rules do not need to have names. Instead they can have *titles* and *descriptions*. Titles and descriptions are passed on to the predicates they produce. They may subsequently be used by the user interface to display advice. Titles and descriptions are however optional.

**Rule Language Specification**

Rules are described using a formal rule language. Figure 5.28 depicts an example featuring two rules described using the rule language.

**Semantics: Illustration by way of an Example**

In the example of Figure 5.28, the rules make use of two predefined types, *object* and *entity.type*. These types are defined as part of the above-mentioned shared ontology. *Objects* refer to graphical objects defined in the SWAN sketchpad. *Entity.types* refer to entities in the Structure Builder. The first rule searches for graphical objects of type "object" in the SWAN builder and tries to match their names with entities in the Structure Builder. Each object found to match an *entity.type* will get a *has.entity* predicate, which is added to the knowledge base. The second rule searches for instances of type *object* that do not hold a predicate relating them to an *entity.type* and who's name attribute is not equal to "unknown". The combination of these two rules effectively maps the objects defined in the SWAN sketchpad onto entities created using the Structure Builder and identifies missing entities in the latter.

**Syntax**

Rules are defined by means of a *Rule statement*. This statement also defines the features it uses. The definition of a used feature consists of two parts, its type and its name, where
5.4. USER INTERFACE IMPLEMENTATION

RuleBase {
    use object, entity_type;

    /* ID: SBC-A-support */
    Rule (object o, entity_type e) {
        if {
            o.name == e.name
        } then {
            has_entity(o);
        }
    }

    /* ID: SBC-A */
    Rule(object o) {
        if {
            o.name != "unknown" and !has_entity(o)
        } then {
            advised(o);
            title {
                Add the object $o.name to the structural model
            }
            explanation {
                You have drawn the Object $o.name in the SWAN Sketchpad. Maybe you would like to add the object as an entity to the entity hierarchy.
            }
        }
    }
}

Figure 5.28
An Example which makes use of the rule language: two rules in an example rule base.

the name must be unique within the scope of the rule. Multiple feature definitions are separated by commas.

The obligatory structure of a rule consist of an if (conditional) block, containing a statement followed by a then (consequence) block eventually holding attribute assignments and association statements. A then block does not need to have any contents, but rules would be senseless without it.

Rules are grouped into RuleBases. Typically, rule bases are stored in separate files. A rule base consists of one or more rules. Furthermore it includes a use statement declaring all types used within the rule base.

5.4.6 Compiler

The rule base compilation process consists of three steps: parsing, translating and assembling. Figure 5.29 illustrates the complete compilation process. Whenever a source code file is read, it is first parsed into tokens. Tokens are the language specification primitives and each one of them is of a certain type. The parsed tokens are subsequently mapped onto a hierarchy, depending on the type and location of the token within the source file. The resulting hierarchy of tokens is called the abstract syntax tree. For the purpose of discerning the various tokens, the parser uses the language specification. Syntactically incorrect files will be rejected.
The second step consists of interpreting the abstract syntax tree and transforming it into an executable representation. For this purpose, executable rule objects are created, which make apparent the semantical structure and thus enable the semantical check which is part of the interpretation process. Any semantics errors detected at this stage will cause the interpretation process to abort and generate an error message.

Finally, the executable representation is assembled into a rule base instance. Rules are placed into the corresponding structures and the rule base is prepared for execution. At this stage no additional check is needed.

**Inference Engine**

The inference engine takes a rule base and a knowledge base as its input. The rule base holds procedural knowledge describing the mechanics of the inference process, the knowledge base holds the facts, or knowledge, which will be used to infer upon.

**Rule Base** A rule base holds the rules that are used by the reasoning engine to infer new facts to be added to the knowledge base. The RuleBase class makes viable the search and retrieval methods included in the reasoning engine and also keeps track of the state of the applied rules in the reasoning progression. A rule base acts as a rule repository. For each distinct context a different rule base, or combination of rule bases, can be deployed.

**Knowledge Base** The knowledge base consisting of features and predicates, acts as a storage depot for facts (a kind of working memory). The inference engine uses these predicates and features for the rule base to infer upon. New inferred facts are in turn added to the knowledge base. The KnowledgeBase class is organised in a way that makes possible an efficient implementation of the search and retrieval methods and thus assists the reasoning engine in its reasoning process.

Figure 5.30 summarises the whole inference process. It starts off with an initial knowledge base. This initial knowledge base is derived from the internal representation of the model and other components in the modelling environment. The first reasoning step consists of searching for rules that apply to the current knowledge base.
These rules are then executed, eventually creating new knowledge and adding it to the knowledge base. This new knowledge could consist of new predicates or new attribute values. At this stage the whole inference process is repeated from the start based on the new knowledge base. A record of the rules that have been applied is kept so as to prevent the repeated application of the same rules. This loop involving search and rule application will continue until there are no more rules left to apply.

As depicted in Figure 5.28, advices are actually predicates of a certain class. At the end of the reasoning process, the knowledge base is queried for predicates of that class. The producing rule provides them with a description and a title, which are then displayed using the graphical user interface.

5.5 Discussion and Conclusions

This chapter has presented the design of MOBUM, a prototype model-building environment for constructing qualitative models. The design of MOBUM was strongly influenced by the experimental results with HOMER (see Chapter 3). Those results highlighted a number of issues concerning the user interaction as well as issues in supporting the task of the users. These newly gained insights were taken into account in the design of MOBUM. The major differences between HOMER and MOBUM are:

- User Interface and Interaction. The interaction in HOMER varies between builders. In some builders the interaction is via menu and in the other builders, the interaction is via buttons in the builder. MOBUM attempts to offer a more consistent and flexible user interaction. All builders employ the same interaction style. All builders have menus, buttons on a toolbar and drop-down menus. Always, to create or edit a model ingredient the user interacts with specific dialogs.
- The ability to run the simulation and inspect the result. The integration with GarpApplet enables the user to directly run the created model. In the case of HOMER, the user needs to save the model to disk, which, in turn can be run and inspected using a separate program, namely VISIGARP [15].

- User Feedback. The agent-based help provides support to the modeler at different levels: It gives support concerning conceptual knowledge and as it reasons on the knowledge constructed that results in advice about completeness and correctness of the model. The agent-based help system explores and inspects the content of existing model ingredients (in a specific builder or across the whole model), identifying useful actions that could be performed within these model ingredients and presenting them in the form of advice.

- Abstraction tools: SWAN SketchPad and Causal Model Builder. The SWAN sketchpad provides the modeler with a drawing tool for sketching purposes. It enables modelers to make a quick and unconstrained drawing of their ideas. Sketching is viewed as an important (first) step in the model-building process [108]. The Causal Model Builder supports the modeler in formalising and specifying the causal model without being concerned with all other aspects of the model.

This chapter has illustrated through a number of examples how MOBUM differs from HOMER. In fact, the main difference between MOBUM and the HOMER environment is in the number of builders and the help system implemented in MOBUM. Despite the overall User Interface organisation and look-and feel, the builders are functionally similar to the ones in HOMER: they facilitate the construction of similar models. The ToolBox and Causal Builder provide functionalities that do not have a counterpart in HOMER. Additionally, MOBUM facilitates the execution of models, by simulation, and the visualisation of the simulation results as an integrated part of the workbench.

Finally, the design methodology followed in this research was an interactive design cycle which consisted of the stages: Design, Prototype, Evaluate. In order to design a user-friendly tool we first needed to define its user base and domain coverage. This lead us to defining the ambitious specifications of the tool, namely, that it should follow a domain-independent approach and be useful to different types of users, ranging from novices to experts.

Based on the results of a rational task analysis and the list of visualisation principles, a first prototype of a model-building tool was designed. The effects of the tool were investigated by means of an experimental study. This study raised some issues and a series of ideas for improvement which were used in the design of MOBUM. An evaluation of MOBUM was performed and is presented in Chapter 6.