Supporting the Construction of Qualitative Knowledge models

Bessa Machado, V.

Citation for published version (APA):
This chapter discusses the design of tools that support the process of building qualitative models and simulations. Firstly, an analysis of model ingredients that can be used in the construction of qualitative models is presented as well as the subtasks involved in this building process. Secondly, the design of a graphical user interface that supports the model building process is investigated. This involves discussing guidelines for visualising model ingredients and subsequently applying these guidelines to the design of specific tools, as well as taking decisions concerning manipulations and basic support within these tools.

2.1 Introduction

Building a qualitative model is a complex process during which a multitude of aspects have to be managed by the model builder. The details of those aspects partly depend on the qualitative reasoning engine that is used. For the research presented in this thesis the domain independent simulator GARP [16] is used. The output of GARP is a behaviour graph that represents the qualitative distinct states a particular system can manifest. To generate such a graph GARP uses a library of model fragments and one or more scenarios, which have to be provided by the users. Figure 2.1 illustrates the situation.

![Figure 2.1](image_url)

*Figure 2.1*

Building a Qualitative Model of System Behaviour.
The idea behind model fragments is that each fragment represents a general concept relevant to the domain that is being modelled, for instance: a population (ecology), a heat-flow (thermodynamics), or a pressure-area (meteorology) [96]. In order to cover a certain domain all the required model fragments have to be defined. Scenarios, on the other hand, are structural descriptions of the particular systems to be reasoned about. Once a library is in place for a certain domain the simulator can be given all kinds of scenarios from that domain to reason about. Thus, at the most general level the problem of building a qualitative model is to create a set of model fragments and to specify one or more scenarios for a certain domain. The output of the simulator, the behaviour graph, provides feedback to the model builder on the model construction process. The model building activity is completed when for each of the specified scenarios the intended behaviour graph is generated by the reasoning engine. The research goal addressed in this chapter concerns the design of a model building environment that supports a model builder in the construction of such a qualitative model.

But exactly how will such an environment support the model builder? To unravel this we assume that at the highest level it is worthwhile to at least distinguish between two tasks (see Figure 2.2). One task is concerned with what must be captured by the model and possibly includes activities such as identifying, selecting and abstracting the relevant features from a set of systems that exist and manifest behaviour in the real-world. The other task is more concerned with how those features should be represented as a coherent set of model ingredients such that the resulting model can be processed by a particular reasoning engine. The latter is referred to as mapping in Figure 2.2. In short, in order to build a model, a modeller needs to have knowledge about the domain and needs to know how to map this into a particular modelling formalism.

![Figure 2.2](image.png)

Figure 2.2
Building a Model for Qualitative Prediction of Behaviour.
Although the two tasks are related, they are influenced by different aspects and provide different possibilities for support. In a learning situation the *what* task depends on the assignment given to the model builder and on the knowledge the model builder has of the domain for which a model has to be constructed. Variations in these two factors will affect the way in which this task is performed. The *mapping* task, on the other hand, depends much less on a specific assignment and the domain knowledge a learner has. Notice, however, that in order to perform the *mapping* task, knowledge about the qualitative reasoning formalism is necessary. This task is primarily determined by the representational means the reasoning engine facilitates. As the latter refers to an implemented and running piece of software, it is safe to assume that no variations will result from using it. That is, it will be the same for all users regardless of their specific assignments and domain foreknowledge. In other words, the *mapping* task is more constrained than the *what* task and thus a better candidate for providing automated support.

This does not imply that it is easy to provide this support. On the contrary, supporting the mapping is a complex problem to solve as can be illustrated by the fact that Graphical User Interfaces (GUIs) for qualitative model building environments are practically non-existing, even though researchers in this area have expressed the importance and necessity of such tools many times [25]. Qualitative models implement a knowledge capturing approach that is rich in terms of the number of ingredients in such models. Consequently, they are dense webs of highly interrelated ingredients [100]. This complexity makes it difficult to build a GUI that supports the construction of qualitative models. As a result, current practice comes down to 'structured programming' of a model using LISP or PROLOG.

This leads to additional arguments in favour of focusing on the *mapping* task first. One concerns the fact that this task is unavoidable when building a model and that in order to have it performed by learners (who are not LISP or PROLOG programmers) tools must be developed that make it possible to execute this task without requiring any programming knowledge. Another aspect concerns the fact that computer tools usually change the way in which a task is performed [89, 61]. Applying that insight to the situation described here means that executing the *what* task will change depending on the tools developed for the *mapping* part. As a consequence, it does not make sense to work on supporting the former, without sufficient insights concerning the latter. Even more so, because there are almost no research results known with respect to how to support the building of qualitative models in the first place.

Following the scoping discussed above, the task to be supported concerns the construction of a set of model ingredients\(^1\) that can be given to a simulator in order to produce a certain simulation. The support will take the form of a GUI that facilitates a user to perform this task. Two requirements are important in this respect. Firstly, the GUI should prevent the user from making errors during the creation of the model ingredients. That is, the output of the task should always be a set of model ingredients that obey the syntax of the simulator. This means that the output can always be simulated by the simulator. Whether the set of model ingredients also represents a *good* model is a different matter.

\(^1\)The terms model ingredients and model are used to refer to the "input" for the simulator. A model consists of a set of model ingredients.
Evaluating the semantic quality of a model largely depends on the modelling goal that needs to be addressed and is beyond the scope of the analysis presented in this chapter. Secondly, the GUI should visualise the model ingredients as insightfully as possible in order to simplify the task for the user as much as possible. We take the position that any GUI requires some learning on behalf of the user and that the understandability of any GUI varies between users depending on the experience and foreknowledge they have in this respect. Therefore, instead of focusing on different users, the problem to solve concerns the development of a visualisation that is \textit{optimal} in terms of the \textit{characteristics} of the model ingredients and their interrelationships.

To address the goal set out for this chapter the following parts are included. Section 2.2 presents an analysis of the \textit{mapping} task as discussed above. The analysis includes a description of the model ingredients that are used to construct a qualitative model as well as a detailed description of all the tasks that must be employed to create these model ingredients. Next, Section 2.3 presents the conceptual design of a GUI for a model building environment that supports the execution of these tasks. This section consists of three parts. Firstly, general guidelines for visualisation are briefly discussed in Section 2.3.1. Secondly, those guidelines are used to design a visual representation of the model ingredients that are described in the task analysis Section 2.3.2. Finally, Section 2.3.3 presents and discusses global decisions concerning the overall interaction and navigation.

### 2.2 Analysing the Model Building Process

This section presents an analysis of the model building process dealing with the construction of qualitative models. Section 2.2.2 presents the ontology used to represent knowledge about qualitative reasoning. Section 2.2.3 discusses all the tasks that must be performed to create these model ingredients in order to construct a qualitative model.

#### 2.2.1 The Scuba Diving Example

This section exploits a model for the scuba diving system in order to explain the ontology.

Maintaining proper buoyancy is one of the most important things about scuba diving. The buoyancy-control device (BCD) is a piece of scuba equipment that allows divers to adjust the buoyancy in the water. It consists of an inflatable bag that attaches to the scuba tank and then to the diver. The diver can control the flow of air into the bag. The amount of air in the BCD influences the position of the diver. Effectively, a diver “feels heavier” the further he descends. By adding air to the BCD, the buoyancy can be increased in a controlled manner and neutral buoyancy can be maintained, which is the goal while diving. As a diver ascend, the reverse is true. One will gain buoyancy as the suit and any air expand. By controlling the deflation of the BCD, neutral buoyancy is maintained. The flow of air determines the amount of air in the BCD. The complete model for the Scuba Diving system can be found in Appendix A. Figure 2.3 illustrates the scuba diving.
2.2.2.2. ANALYSING THE MODEL BUILDING PROCESS

Figur ee 2.3
Scuba Diving illustration.

2.2.2 Ontology

Ontology in this thesis consists of the vocabulary that is used to represent knowledge in a domain. The vocabulary consists of concepts and relations between these concepts. What is produced using this vocabulary is the representation of a model of some system. Models are composed of a series of model constructs, which represent fragments of the complete system model. These model constructs are assembled from a set of building blocks, which model certain aspects of the system's structure and behaviour. Together, model constructs and building blocks are what we refer to as model ingredients. This section describes the different types of model ingredients that can be used when qualitatively modelling a system. Figure 2.4 gives an overview of these model ingredients and their relationships. The section ends with a discussion of the results produced by the simulator, i.e. the state graph.

Building Blocks: System Structure

Modelling the structure of a physical system in the real world implies representing the objects that a certain system consists of. In GARP, three model ingredients are employed to describe the structure of a system: entities, structural relations between entities and attributes of entities.

Entities represent physical objects or conceptualisations which are part of the real world problem domain. Entities are organised by means of an isa hierarchy. The entity hierarchy used in the scuba diving domain is shown in Figure 2.5.

Structural Relations model how the system's entities relate to each other. For instance, the following structural relations should be part of the Scuba Diving example: diver dives in a river, BCD contains air and BCD connected to a diver.

Attributes represent static properties of entities. That is, features that usually do not change as part of the systems behaviour. When modelling attributes, the set of values that an attribute may have must also be specified. In the Scuba Diving system, colour is specified as an attribute for BCD, with possible values black and red.
Building Blocks: Behaviour

Modelling the behaviour of a system is about representing what makes change possible in a system or, in other words, representing how a system changes over time. Model ingredients such as quantities, quantity values, derivatives and dependencies are the ways of representing behavioural properties.

Quantities are behavioural properties of entities that may change over time. Quantities have values, and possible values for a quantity are represented in a quantity space. As we are working with qualitative models a quantity space consists of a set of ordered points and intervals that represent an abstraction of a quantitative range. Associated with the entities for the Scuba Diving system, the following quantities are modelled. For the BCD, Flow, for the diver Position, and for air (in the BCD) the quantity Amount. Moreover, surface and bottom are point values in the quantity space for the quantity Position of the diver in the river and middle is the interval between these two point values. Additionally, quantities have a derivative. The derivative represents the direction in which the quantity value changes. The magnitude of the change is abstracted and only the sign of the change is captured. Thus,
the derivative can assume the values -, 0, + representing respectively *decreasing*, *steady* and *increasing* quantity values.

**Causal dependencies** are essential in predicting system behaviour. They specify relationships between quantities. Causal dependencies represent changes on one quantity caused by another quantity. Three types of causal dependencies are defined within the ontology: *Influences*, *Proportionalities* and *Correspondences*. An *influence* (either negative or positive) specifies that the value of a quantity $Q_1$ determines the derivative of quantity $Q_2$. For instance, in the Scuba Diving system, the *Flow* (negative/zero/positive) in the BCD changes the derivative of the *amount of air* (decreasing, steady, increasing) in the BCD. A *proportionality* can be also negative or positive. A positive proportionality between $Q_1$ and $Q_2$ implies that a change in $Q_1$ causes a change in $Q_2$ in the same direction (If $Q_1$ increases, $Q_2$ increases as well). In the case of a negative proportionality, a change in $Q_1$ causes a change in $Q_2$ in the opposite direction (If $Q_1$ increases, $Q_2$ decreases). For example, the *amount of air* is related through a positive proportionality to the *position* of the diver on a river. If only the amount of air increases (decreases), i.e. all other factors are invariant, the position will increase (decrease) as well. In this case the diver will move towards the surface (or go down).

A *Correspondence* gives the means of mapping values from the quantity space of one quantity onto values in the quantity space of another quantity. Correspondences can be of different types. An *undirected correspondence* is used to express a two-way relation between the quantity spaces of two quantities, meaning that for every qualitative value of $Q_1$, there is a corresponding value of $Q_2$ and vice versa. Specifically, it denotes that if either $Q_1$ or $Q_2$ is known the other can be derived. In a *directed correspondence*, the relation only holds in one way, which means that if $Q_1$ is known, $Q_2$ can be derived but not vice-versa. Correspondences may also exist between specific quantity values, as opposed to the whole quantity space. They are similar to the correspondences above and are known as *undirected value correspondence* and *direct value correspondence*. In the Scuba Diving, the value *minimum* of the quantity *Amount* (of Air in the BCD) corresponds to the value *bottom* of quantity *Position* (of the diver). This is a directed correspondence because the reverse is not always true, that is a BCD may still have more than minimum air while the
dive is at the bottom.

**Non-causal dependencies** are the means of representing constraints on or between quantities. In GARP, the non-causal dependencies are the (in)equalities\(^2\). Inequalities can be used for instance to describe how the point value of one quantity relates to the point value of another quantity, but also to describe specific types of dependencies between two quantities which may subsequently be used in computing values of other quantities. By using inequalities we can represent a partial or complete order between the values and derivatives of different quantities. Standard inequality relations are \( <, \leq, =, \geq \) and \( > \). Each side of an inequality may be replaced by an *addition* or a *subtraction* by using the 'plus' and 'min' operators. For example, \( \partial Q_1 = \partial Q_2 + \partial Q_3 \) or \( Q_4 > Q_5 = Q_6 \). In summary, the following inequalities types exist:

- \( Q_1 \leftrightarrow Q_2 \): Inequalities between two quantities.
- \( Q_1v \leftrightarrow Q_2v \): Inequalities between quantity values are used to relate point values from different quantity spaces.
- \( Q \leftrightarrow V \): An inequality between a quantity and a value can either define a specific value for the quantity (an equality) or a set of possible values (an inequality). For example, taking a quantity space with the values *zero*, *plus*, *maximum*, if the value of a quantity is defined as *equal* to zero then the quantity has that specific value. On the other hand, if the value is defined as *greater-than* zero, then the quantity value is either *plus* or *maximum*.
- The same inequalities apply for derivatives. Inequalities can be defined between two derivatives, \( \partial Q_1 \leftrightarrow \partial Q_2 \), between two derivatives values, \( \partial Q_1v \leftrightarrow \partial Q_2v \), and between a quantity and its derivative value, \( \partial Q_1 \leftrightarrow \partial v \).

In the Scuba Diving example, when a diver inflates the BCD, the *Flow* of air in the BCD is greater-than *zero*.

**Model Constructs**

Model constructs are the model ingredients that the simulator uses for behaviour prediction. In the following the two types of model constructs are described: model fragments and scenarios.

**Model Fragments** The structural and behavioural building blocks discussed above are used to compose *model fragments*. Model fragments encode knowledge about the behaviour of "subsystems" and thereby enable us to organise the system's behavioural properties into small understandable blocks. A model fragment is organised into two parts: the *conditions* and the *consequences*. Conditions consist of the system's structure (entities, attributes and structural relations) but may also include

\(^2\)We use the term inequality throughout this thesis to refer to both inequality and equality relations.
some knowledge about the system's behaviour in the form of quantities, inequalities and other model fragments. Conditions must be true for the model fragment to hold. The consequences are the direct implications of the model fragment being active. Besides the constructs allowed in the conditions, the consequences may additionally include causal dependencies. Different model fragments are employed in the Scuba Diving system. In the following a model fragment, 'A scuba diver in a river', is presented. The model fragment specifies the position of a diver in a river and the amount of air. The amount is positively proportional to the position: if the amount increases, the position will increase. The model fragments Diver and BCD are conditional for this model fragment. Below, the complete model fragment for Scuba Diver in River is presented.

### Model Fragment

**IF**
- a model fragment Diver applies
- and model fragment BCD applies
- and river River exists
- and the relation dives-in(Diver, River) holds
- and the relation connected-to(BCD, Diver) holds

**THEN**
- Air has an amount Amount
- and Diver has a position Position
- and Amount is positively proportional to Position

Model Fragments are hierarchically organised (one model fragment may be a specialisation of another) thereby supporting inheritance. Model fragments can be of different types. At the top level of the hierarchy four basic types are used in GARP:

- **Description view** describes functional and behavioural aspects of a single entity (e.g. the model fragments 'Diver' and 'BCD' conditionals in the example above.
- **Composition view** is used for aggregations of entities, the Scuba Diver in River example given above.
- **Process** represents interaction between entities usually triggered by an inequality. Our Scuba Diving does not use processes. However, a typical example of a process is heat flow, for example, two objects of different temperatures, see [43].
- **Agent** view focuses on external effects on a system. For instance, a diver's actions of inflating and deflating the BCD are examples of 'agent views'.

### Scenarios

A scenario is an initial description of a system at a specific time. It defines the starting point of a behaviour simulation. A scenario description consists of both structural and behavioural building blocks. The system structure is represented by the three types of structural model ingredients discussed above: entities, attributes and structural relations. The behavioural properties refer to the initial values of the system as well as to inequalities that set boundaries on the system's behaviour. While model fragments are the representation of the generic domain knowledge, scenarios describe instantiated knowledge about a specific situation. A scenario for
the Scuba Diving system is illustrated in Figure 2.6. The scenario specifies that the Diver is at the bottom with the air in the BCD in its lowest value. The Flow is positive.

![Diagram of Scuba Diving scenario](image)

Figure 2.6
Example of a scenario of Scuba Diving.

**Behaviour Graph: Output of the simulator**

GARP takes as input a scenario and generates a graph of qualitative distinct behaviours for the situation described in that scenario. A state refers to a period in which all values, derivatives and inequalities remain the same. A state transition occurs when inequalities between quantities change or when quantities get a different value from their quantity space. Such changes are the result of influences introduced by model fragments (of type 'process' and 'agent'). These influences propagate via proportionalities and affect the derivatives of quantities. In other words, they change the quantities and this is modelled as derivatives being positive or negative. In this way, the state graph shows an overview of the progress of the simulation ordered in time and represents the potential behaviour of the system. In Figure 2.7 all generated states are depicted given the scenario example above.

Each possible path through the state transition graph represents a possible behaviour over time. In state 1 the DIVER starts inflating the BCD, causing the amount of AIR to increase and making the DIVER wanting to go up (position increases). The behaviour in state 1 terminates into state 2. In this state the DIVER is in between the bottom and the surface (position has value: middle) and the BCD is partially filled with AIR (amount has value: plus). A set of possible successor states are then possible: the DIVER reaching the surface before the BDC is fully inflated (state 6) or reaching it at the moment the BCD is fully inflated (state 5). Alternatively the DIVER may not reach the surface, even though the BCD is fully inflated (state 4), or the DIVER may stay below the surface while not having the BCD fully inflated. New actions, in terms of inflating and deflating the BCD may cause these states to terminate into other states again.

2.2.3 **Rational Task Analysis**

This section uses the task analysis technique as a means for determining the way in which models are built, thereby breaking down the model building process into its constituent
2.2. ANALYSING THE MODEL BUILDING PROCESS

Figure 2.7
Behavioural States for the Scuba Diving Example.

tasks. The focus is on rational task analysis, that is studying the subject matter and specifying the process that is presumed to be involved in doing the task [3].

We chose to use in this chapter UML activity diagrams [14] for structuring and outlining the task analysis. By opting for activity diagrams, the analysis is centered around the information needed to accomplish a specific task and at the same time focuses on the actions the user must perform as well as on the constraints imposed on these actions. Activity diagrams offer a definitive answer as to the order in which the performance of a task might occur.

Figure 2.8 summarises UML activity diagram notations.

Figure 2.8
UML Activity Diagram notation.

An action state (rounded rectangle) represents an activity or process that is performed. In our case, the activities that matter are the ones that transform or create a model ingredient by manipulating, restructuring or relating it to other model ingredients. The arrows represent transitions between activities, modelling the flow order between the
various activities. The thick bar represent the start and end of potentially parallel processes/activities. An inward directed arrow represents that the action state uses the object, if the arrow is pointing towards the action state. An outward arrow represents that the action creates or influences the object. An Object (rectangle) in a state represents instances of a specified object in a specified state, in our case model constructs and building blocks. Action states have access to the objects to generate, store or transform data. The text on the arrows represent conditions that must be fulfilled to proceed along the transition.

Main Tasks

The final goal of model building is to arrive at a complete model that can be simulated, and thus be used to predict a system's behaviour with the aid of a simulator.

In Figure 2.9 we have decomposed the creation of a simulation model into its main subtasks. The user will employ a set of Building Blocks in order to construct a model. S/he will then assemble these building blocks into dedicated Model Constructs, which will be used by the simulator as input for predicting a system's behaviour. This creation of building blocks and model constructs must obey a set of rules. A verification step is therefore required before considering the model to be syntactically correct. The model will be created only when no errors are present. The arrow connecting the model to the task of creating a model construct represents the dependencies between the various building blocks of the complete model. It states that, when creating new model constructs, it may be necessary to access the previously created knowledge.

In the following sections the two main tasks presented above are decomposed into subtasks until such a fine-grained level is attained that permits us to fully understand all the steps needed to arrive at a full qualitative model.
2.2. ANALYSING THE MODEL BUILDING PROCESS

Creating Building Blocks

Figure 2.10\(^3\) shows the tasks that are required for the creation of building blocks and for the creation of model constructs. Looking at diagram 1, one may notice that only the task of 'Create Quantity' has a constraint: at least one quantity space must have been created beforehand because for all quantities a quantity space must be specified. Furthermore for all subtasks in the diagram a verification is necessary before considering the subtask completed. Diagram 2 shows the subtasks of creating model constructs. Similar to the subtasks of creating building blocks, a verification step is necessary for all subtasks.

1. Create Building Blocks

2. Create Model Constructs

Figure 2.10
Main tasks in a more detailed view.

In the following paragraphs we descend into one more level of detail, giving activity diagrams and an accompanying textual description for each of the subtasks. Notice that in Figure 2.10, we have assigned number "1" to the diagram relative to 'Create Building Blocks' and number "2" to the diagram corresponding to 'Create Model Constructs'. We will follow the same numbering convention here. As such, the diagram for the "Create Entity" task is numbered 1.1 due to the fact that it is associated with the previous diagram 1 and it is the first task to be described. Similarly, the diagram for the "Create Model Fragment" task is numbered 2.1 and so forth.

When creating an entity, diagram 1.1 in Figure 2.11, a series of distinct tasks take place. As mentioned before, entities are organised into a subtype hierarchy. The user must therefore always specify the entity's super type. Every entity should have a unique name.

\(^3\)We use the term 'Figure' to refer to the full picture. The term 'diagram' is used to refer to one of the subfigures in the complete Figure. For example, Figure 2.10 contains two diagrams: diagram 1 and diagram 2 which refer to the tasks of 'Create building blocks' and 'Create model constructs', respectively.
as its identifier. Also a short description stating the entity’s meaning\(^4\) may be associated with it. The last step consists of validating the newly created object and adding it to the subtype hierarchy if all validity tests succeed. In this specific case an entity is valid if no other entity with the same identifier exists and a subtype relation is present. Thus, the verification step prevents the specification of cyclic hierarchical relations, (for instance, a super type of an entity is not one of its descendants) as well as the specification of identical subtypes.

Diagram 1.2, Figure 2.11, shows the tasks for creating an attribute. The user must give a unique identifier and specify the values of the attribute (for instance, an attribute of a container can be openness and its possible values: open and closed). In diagram 1.3, Figure 2.11, the task of creating a structural relation consists mainly of giving it a name. Recall that structural relations are the means by which we relate two entities. At this stage, however, we are discussing the creation of the basic building blocks. Structural relations only come in to play in the context of building model constructs. Only at that stage, will the user define the entities involved in the structural relation just created. It is therefore, that no reference is made to entities in the diagram 1.3 of Figure 2.11.

![Diagram 1.1 Create Entity, 1.2 Create Attribute, 1.3 Create Structural Relation](image)

Figure 2.11
Creating an entity, attribute and structural relation.

Diagram 1.4, Figure 2.12, shows the creation of a quantity space. The user names the new quantity space and specifies the set of values constituting the quantity space. Besides verifying the uniqueness of the given name, the verification step checks whether the given sequence of values is valid. A sequence of values is valid if it follows the rule of having alternating unique points and intervals. The last task within creating building blocks, Diagram 1.5, Figure 2.12, is the creation of a quantity. Besides giving it a name, the user has to select a previously created quantity space for the new quantity.

\(^4\)For example, a description of the real world "thing" that the entity refers to.
In summary, every building block described above must have a unique identifier and may include a short description. Also, new building blocks must always be verified before being stored. The three subtasks of giving a name, giving a description and validating the data are therefore common to all diagrams.

Making Model Constructs

Model constructs are either model fragments or scenarios. This section analyses the creation of these model constructs. Figure 2.13 includes two activity diagrams representing all subtasks that may possibly be carried out in the context of creating model fragments or scenarios. As creating a scenario is a subset of the more complex task of creating a model fragment it is sufficient to describe only the latter.

Two major task categories can be identified: Adding Building Blocks and Adding Dependencies. The first category consists of using the previously created building blocks to compose model constructs. The second category includes the addition of the various types of dependencies (see Section 2.2.2) which apply to model constructs. These categories will be described in the following paragraphs.

Adding Building Blocks

Figure 2.14, diagram 2.1.3, represents the task of adding an entity to a model fragment. In order to add an entity to a model fragment, the following subtasks are needed: 1) selection of its generic type from the (is-a) hierarchy of entities, 2) providing a unique name to the entity and 3) specifying whether the entity is either a condition or a consequence
within the present model fragment. As in the previously described tasks of creating building blocks, the last step consists of validating the entity and adding it to the model fragment object if all validity tests succeed. In this case, the verification consists of verifying whether the entity name is unique. Figure 2.14, diagram 2.1.4, shows the task of adding an attribute to the model fragment. To do so, the user needs to first select an entity (from the present model fragment) to which the attribute is given. Then the user must specify the attribute’s value. Recall, that when creating a new attribute, the user needs to specify the possible values for the new attribute. In the present context, the user must set the attribute’s value to one of those predefined values. E.g., when adding the attribute openness to the entity container, its value must be set to either open or closed.

When adding a structural relation to a model fragment the user is relating two entities.

---

5 Scenarios do not have this distinction. The building blocks that can be specified in a scenario are the same as for the conditions of a model fragment.
2.2. ANALYSING THE MODEL BUILDING PROCESS

2.1.3 Add Entity

2.1.4 Add Attribute

2.1.5 Add Structural Relation

Figure 2.14
Add Entity, Attribute and Relation to a Model Fragment.

Thus, the user’s task consists of selecting these two entities, (Figure 2.14, diagram 2.1.5) and applying one of the previously created relations. Here, the verification consists of verifying if this relation has already been used in combination with the present pair of entities. Consider, for instance, that a structural relation contains has already been added to the model fragment relating the entities container A and water. Imagine now, that the user wants to add a structural relation of the same type between the entities Container B and gas. Since they are not the same entities, this would be a valid operation.

Figure 2.15, diagram 2.1.6, describes the subtask of adding a quantity to a model fragment that is similar to the ones presented above. This subtask consists mainly of selecting the entity that the quantity belongs to. Naturally, the quantity must be unique to that entity. Notice that, when the quantities are created they are already assigned a unique
Add a Quantity and Quantity Value to a Model Fragment.

Since a quantity is always associated to a unique quantity space, the quantity’s specific value must be chosen from the quantity space’s set of possible values. Within one model fragment (or scenario) a quantity may not be attributed more than one specific value.

Adding Dependencies

Two categories of dependencies exist: non-causal and causal dependencies. Figure 2.16 shows the diagrams representing the three different subtasks of adding value-quantity, quantity-quantity and value-value inequalities. In order to *Add a value inequality*, diagram 2.1.8, the model builder must select the quantity that the inequality applies to, then s/he must choose a specific value from within the quantity’s quantity space and finally specify the type of inequality. The next diagram in the figure, diagram 2.1.9, shows the task of *adding an inequality between two different quantities*. This task consists of selecting the two quantities, referred to as LHS and RHS quantity, as well as selecting the type of inequality that holds between them. Figure 2.16, diagram 2.1.10, depicts the subtask of *adding an inequality between two quantity values*. First the LHS and RHS quantities must be selected. Second, the appropriate values must be selected from the quantities’ quantity spaces and finally, the type of inequality must be chosen. As inequalities can only exist between point values, in the verification step it must be checked whether the selected values are point values.
2.2. **ANALYSING THE MODEL BUILDING PROCESS**

2.1.8 Add Value Inequality

2.1.9 Add Quantity/Quantity Inequality

2.1.10 Add Value/Value Inequality

---

**Figure 2.16**  
Adding Inequalities to a Model Fragment.

**Causal Dependencies**

While the non-causal dependencies described above can be used as conditions or as consequences in a model fragment, causal dependencies can only be specified as consequences in model fragments. Also causal dependencies can not be used in scenarios. Diagrams 2.1.11 and 2.1.13 in Figure 2.17, are almost identical, the only difference being that the first one refers to the creation of influences\(^6\), while the second is about specifying correspondences. In both cases, the model builder initially must select the influencing/corresponding LHS quantity and then the affected/corresponding RHS quantity. After that, the model builder must select the type of influence or correspondence that holds between the two selected quantities. Diagram 2.1.12, Figure 2.17, shows the subtask of

---

\(^6\)There are two kinds of influence: positive and negative.
Adding a value correspondence to a model fragment. A value correspondence involves two quantity values. The steps needed to complete this task consist of the following: 1) Select the LHS and RHS quantities, 2) from their quantity spaces, select the involved values, and 3) specify the type of correspondence.

2.1.11 Add Influence

2.1.12 Add Correspondence

2.1.13 Add Value-Correspondence

Figure 2.17
Adding an Influence and Correspondences to a Model Fragment.

Summarising the task analysis

Figure 2.18 summarises the result of the task analysis. It suggests the existence of seven main modelling activities. They are the creation of entities, attributes, structural relations, quantity spaces, quantities, scenarios and model fragments.
2.2. ANALYSING THE MODEL BUILDING PROCESS

Notice that, in order to perform the task of creating a quantity, at least one quantity space must have been created. This is due to the fact that a quantity must always be associated with a quantity space. Additionally, to initiate the last two tasks (creation of scenarios and model fragments), the other five modelling activities must have been previously concluded. Demanding that the generic definition of the building blocks be concluded before defining the model constructs, makes the overall task more manageable and consistent. Furthermore it forces the model builder to focus on the specific model building activities. It may also happen that the same building blocks play a role in several
different model construct definitions. Thus, creating building blocks in advance will make the model construct definitions more efficient by allowing the task to be carried out mainly by selecting the right building blocks.

From the diagram in Figure 2.18, the question arises how a user interface can support the model builder in executing the (sub)tasks. The seven main separate modelling activities, mentioned above, suggest that the design of a model building environment should include seven workspaces, each of them supporting the execution of a single main task. These seven dedicated windows are hereafter called builders. The design of the builder is further discussed in the following sections.

2.3 Designing Modelling Tools

How to design a graphical interface that supports a model builder in executing the tasks as identified in the task analysis? To answer that question, two aspects need to be addressed. The first one focuses on how to graphically organise the model ingredients on the screen. The second one focuses on the kind of manipulations a model builder can perform with that structure. Those two issues are addressed in three sections. Section 2.3.1 discusses visualisation of model ingredients in general. Section 2.3.2 refines that discussion by applying the general principles to the specific model ingredients as defined before in Section 2.2.2. Finally, Section 2.3.3 focuses on the requirements for interaction and navigation.

2.3.1 Guidelines for Visualising Model Ingredients

One criterion for deciding how to visualise and organise the model ingredients on the screen concerns the information that needs to be communicated to a model builder when placing a particular ingredient on the screen7. A model ingredient typically has a (model-)type and a (class) name that must be communicated to the model builder. The model-type refers to the primitives of the Qualitative Reasoning vocabulary, while the name refers to a class of a specific model-type. The model builder should be able to identify and modify those on the screen, for example, the 'quantity' 'temperature', being 'model-type' and 'class-name' respectively. Notice that the set of model-types is fixed, because it is defined in the modelling language, see Section 2.2.2 for an overview. The class-name, on the other hand, refers to a specific class of a model-type created by a model builder and therefore differs depending on the modelling activities carried out by the model builder. Allowing maximum freedom for the model builder requires that the latter is visualised on the screen using text labels provided by the model builder. In fact, a parsimonious way to visualise an ingredient would be to only use the text label as the identifier on the screen, for instance the name of the ingredient, in the example 'temperature'. But, in the case of multiple ingredients of different model-types, this approach is insufficient. By just showing the names it will be difficult, if not impossible, for a model builder to identity

---

7In this text we use the words 'screen' and 'window' as synonyms. Both refer to a selected area on a computer display that is designed to facilitate a specific kind of interaction with a model builder. De facto, based on the results of the task analysis seven areas, called builders, should support the construction of the main tasks. Usually a window is marked by clear boundaries.
the different model-types. Therefore a second visual cue is usually needed to highlight the model-type of an ingredient. This can, for instance, be achieved by colour-coding the ingredient model-types. Thus, the class-names of different model-types would have different colours. Another approach would be to use a second text-label that explicitly states the model-type. However, a more common approach to communicate types is to use icons [29, 59]. The alternative options are shown in Figure 2.19. In summary, we believe that the better way to communicate the model-type and the class-name is by using icons and text labels, respectively.

![Figure 2.19](image)

Alternative approaches for visualising model-types and classes.

However, our situation is slightly more complex than discussed above. Some model ingredients represent three aspects instead of two, namely model-type, classes, and instances. Examples of ingredient model-type are the notions such as entity and quantity. This set of modelling ingredients is determined by the modelling language and is thus fixed from a model building point of view. A class refers to a specific creation of those ingredient model-types in the context of a domain. Examples of quantities are temperature and volume in the case of physics, or size and birth-rate in the case of ecology. Finally, specific occurrences of those quantities when modelling a system from such a domain are referred to as instances. Examples are the specific temperature of a certain object, or the size of a specific ant population, see also Figure 2.20.

Because the model construction environment should be domain independent, icons can only be used to refer to 'model-types'. As mentioned before, the set of ingredient model-types is fixed. Classes and instances on the other hand are not fixed. It varies depending on the domain and system for which a model is constructed and the specific viewpoints taken by the model builder. Allowing maximum freedom for the model builder in this respect requires visualisation on the screen using names given by the model builder. Another aspect that would complicate an icon-based visualisation of the classes is that intuitive and insightful icons may not always be available (how to visualise, for instance

---

8In fact, we should distinguish between occurrence and instances. Occurrences refer to building blocks reused to create a model fragment, which is a construct. Instances refer to building blocks used to create scenarios. Model fragments represent generic knowledge, hence the term occurrence. Occurrences become instantiated when a model fragment applies to a scenario. However, during model building, occurrences and instances (as defined here) do not appear in one workspace. Therefore we can treat them in a same way, that is, distinguish them in a workspace from model-types and class-names.
a 'volume'?). Thus, if a model ingredient represents three aspects, an icon can be used to communicate its model-type whereas text labels must be used to communicate the information about class and instance.

A second criterion that can be used to visualise and organise model ingredients on the screen exploits the relationships between model ingredients. Model ingredients differ in terms of how they relate to each other. In Section 2.2.2 a distinction was made between building blocks and constructs, the latter being specific organisations of sets of building blocks. But also within the set of building blocks certain ingredients, such as dependencies, are meant to represent relationships between model ingredients. For the discussion presented in this chapter, the following organisations between model ingredients should be pointed out: unrelated, binary and structure. Unrelated means that an ingredient has no relationship with other ingredients. Binary means that there is a particular kind of relationship between two ingredients. Structure refers to the case in which a set of ingredients are related according to a particular format. Typically two extreme versions of the latter exist. The simplest case, in which all the ingredients are of a single model-type and only one relationship type is used to relate them, and the most complex case, in which a variety of model ingredients of different types are connected using multiple types of relationships. During a model building process the relationship between ingredients can change. For instance, although in a complete model all ingredients are always related (by definition), it may be the case that during the model building process they are, at least for a certain period, unrelated. This may affect the way ingredients need to be visualised.

Combining the two criteria presented before with the relationship issue above leads to a number of considerations. One concerns the situation in which all the ingredients in a certain window are of the same model-type and unrelated. In this case the name of the window may refer to the model-type that is shown and thus the icons highlighting the model-type can be omitted. Although, formally this is correct, there might still be reasons for including the icon showing the model-type. For instance, because it is consistent with the overall look and feel of the interface to always show a certain model ingredient by means of its icon, model-type, and its class-name. Another approach with unrelated items would be to present them as a list. This may be better than a graphical format because it takes less space and provides easier access, for instance by being able to use alphabetic
ordering.

A second issue concerns a set of binary relationships that have a model-type but not a class or an instance. Take the following expression: $X > Y$. On the screen it should be pointed out that $X$ and $Y$ are quantities and that they have a certain name, for example \textit{volume 1} and \textit{volume 2}. But the ‘$>$’ relation may not need any further clarification, beside the ‘$>$’ icon, because it only requires a model-type. The relation does not exist (or make sense) without it being connected to two other model ingredients. Therefore, the relationship does not need a specific name to distinguish it from other relationships of that model-type. By definition, such binary relationships are uniquely identified by the ingredients they relate. The alternative can also happen: some relationships only require the model builder given name to be visualised because the model-type can be determined solely by the context. E.g. structural relations between entities are the only kind of relationship that can exist between these entities. Therefore, they are unique and it is sufficient to show only the model builder given name in the screen.

A third and related issue concerns a window in which a structure is shown that uses only one type of relationship between the ingredients. In such a situation a simple line (that is, without any deliberate meaning attached to it) between the ingredients that have the relationship may suffice. Take for example a subtype hierarchy. If the window that displays the ingredients has an insightful name, then there is usually no need to explicitly denote each relation between pairs of ingredients in the hierarchy as being a subtype relationship. The fact that such a figure represents a hierarchy already implies that all the relationships must be subtype relations. Moreover, it is probably more insightful to not repeatedly denote the relationship, because it clutters the screen with information that can be considered 'default knowledge'.

A fourth issue to be addressed concerns a structure with multiple relationship types and the difference between 'implicit' and 'explicit' relationships. In the examples above the relationships 'subtype' and ' $>$' are explicit in the sense that they are primitives in the vocabulary that is used for building models (see Section 2.2.2). However, some other relationships between model ingredients are implicit, that is they exist but are not explicitly referred to by a model primitive in the language. Assigning a 'quantity' to an 'entity' is an example of such a relationship. There is no relation type in the model language that represents this relationship (e.g., part-of, has-attribute or belongs-to), instead the quantity directly relates to the entity. Presented visually on the screen does however require some representation of the notion 'belongs-to', otherwise the model builder will not be able to identify which quantities belong to a certain entity. At least three solutions can be pointed out. One would be to use some kind of 'visual container' to represent the entity, for instance a circle or a square, and to display all the quantities belonging to the entity in that container. This approach is used in VisiGarp [15]. Another approach would be to view quantities as \textit{relationships} between entities and values. Visually this could be represented by a special kind of line, possible augmented with a specific icon, between the entity and the value (for that quantity) (see also Figure 2.21).

A possible drawback of this approach is the representation of dependencies between quantities (or in general: other relationships relating to this model ingredient). Dependencies seem to require quantities to be represented as icons, so that a dependency can be
Figure 2.21
Approaches in visualising entities, quantities and values.

placed as a line connecting two quantity icons (see also the discussion on binary relationships above). A third approach would be to represent the 'belongs-to' relations by means of a simple line connecting the icons involved. Whether the line needs to be augmented with a specific icon to denote the relationship type depends on the other relationship types that need to be visualised. In general, one relationship type can be allowed to represent a default value if all other relationship types have a uniquely identifiable visualisation.

Summarising, visual presentation of model ingredients on the screen primarily depends on the information it needs to convey to the model builder. Icons are preferred for the parts that are fixed and cannot change. This set is defined by the modelling language. Text labels are preferred for the parts that depend on the model builder. This set mainly refers to the things a model builder wants to capture (represent) in a model. Some model ingredients deliberately have the role of connecting other model ingredients. This connectivity feature between ingredients further refines the visual representation needs. Typically, one relationship can be represented as the default in a particular window and therefore be minimally visualised, for instance by a thin line connecting the related things. Next, binary relationship may not need a unique identifier, next to their type identifier, because they are by definition defined by the two things they relate. Finally, some relationships between model ingredients may be implicit in the modelling language, but may need an explicit visual representation on the screen.

2.3.2 Visualising Model Ingredients

Section 2.2.3 presents a detailed analysis of the task of constructing qualitative models and in Section 2.3.1 the criteria for visualising model ingredients are discussed. In this section, we use the results from the above-mentioned sections in order to discuss how to visualise the model ingredients. In the discussion that follows, ways of implementing the model builder interface of a model building environment are presented. This is strongly influenced by the activity subdivision presented in Section 2.2.3 (see also Figure 2.18) and therefore, a different builder will be associated with each of the seven main modelling activities.
2.3. DESIGNING MODELLING TOOLS

Entities

Entities represent the objects/concepts of the real world domain. Entities are hierarchically organised. In effect, this is a structure because all entities have a relationship with at least one other entity. But the structure includes only two model ingredient types: entities and subtype relationships. For the entities the model builder provides a new class-name for each subtype added to the hierarchy. The subtype relation, on the other hand, does not have a model builder given name. Thus, in total three aspects must be communicated to the model builder: model-type (entity), name (for an class entity given by a model builder) and type (is-a relation). Placing this information in a table is not a very good option, although in principle possible. Mainly, because in a table it will be difficult to get an easy to read overview of all the relationships in the structure. It would, for instance, be difficult to quickly see whether two entities have a subtype relation, particularly when the two are indirectly related via a chain of intermediate entities.

Figure 2.22 shows possible ways of graphically representing the entity subtype structure. Figure 2.22a uses a minimal approach. It shows the class-names given by the model builder (but not the entity model-type icon) and a connecting line to visualise the relationships between the entities. To what model ingredient type the model builder given names belong, must be inferred from the name of the window (shown in the title bar). Also the meaning of the connecting line must be inferred from the window title bar. But notice that strictly speaking a model builder may still not know whether the cube or the text in the title bar refers to either the model builder-given class name or the connecting subtype line. The layout in the title bar does not correspond insightfully to the layout in the window.

Figure 2.22
Hierarchy view.

The approach taken in Figure 2.22b tries to visualise all the three aspects in the window, by using icons for model-types\(^9\) and text-labels. Although this approach explicitly communicates all the important aspects, it is arguably sub-optimal. For one thing, the visualisation is somewhat cluttered, which will become more problematic when the number of elements in the hierarchy increases. Furthermore, all the relationships in the window are of only one type. It is probably better to use a default line to represent them. The same seems to hold for the entity icon: all the given names refer to entity types and therefore

\(^9\)Actually, following the guidelines discussed in Section 2.3.1 the is-a relationship identifier should also be an icon. But the is-a is difficult to represent as an icon so it is done as a text label.
the entity type icon can be left out. Which would bring the situation back to Figure 2.22a. However, there is a difference in the use of the subtype relationship compared to the entity type. The latter will also be used in the visualisation of model fragments and thus appear in the context of many other ingredient types. In that context the model-type icons become essential and cannot be left out. This is not the case for the subtype relationship between entities. In fact, this relationship will not be shown in any other window. In order to enhance the consistency for the whole environment it would be good to always show the same information for a particular model ingredient. Following this guideline, the model-type icon for the entity should be preserved whereas the is-a model-type icon can be removed. This approach is shown in Figure 2.22c. The title bar shows the relationship type that is represented as lines between the entities.

Apart from the specific visualisation chosen, the structure always remains a tree. The layout of a tree is common practice in Artificial Intelligence as in many other areas of Computer Science [105]. The typical organisations are shown in Figure 2.23 and are generally well understood by model builders [109].

Attributes

Attributes represent the static features of entities. They hold a set of possible values. The information that must be communicated to the model builder consists of the model-type (attribute), the attribute's given class-name, the model-type value and the values' given names. As before, the types can be visualised by icons, whereas the model builder given names will be text labels. However, the value model-type icon can be left out because a specific value is always associated to the attribute it belongs to. It never appears by itself in any other context. Thus, text labels are sufficient to show the information about the values.

Figure 2.24a presents a version for visualising this model ingredient. It shows a line connecting the attribute's name to its values. This representation is acceptable for limited set of attributes but when the number of attributes increases this form of visualisation may become cluttered. It will then also be difficult to access a specific item. The second example, Figure 2.24b, shows an alternative way of representing the attributes. It takes
advantage of the fact that there is no relation between different attributes. And therefore, applying the guidelines from Section 2.3.1, a simple one dimensional list suffices to compactly represent the model's attribute. The possible values of an attribute will only be displayed upon request, by selecting the attribute in the list. This approach takes less space and provides easier access by allowing the data to be alphabetically organised. Notice that, once this model ingredient is used in other contexts, both approaches keep the attribute type icon on the screen in order to maintain consistency in the whole interface.

![Attributes view](image)

**Figure 2.24**
Attributes view.

### Structural Relations

Structural relations are the means for relating entities. Given that structural relations are the only model ingredient that can relate two entities, the model-type identifier is no longer necessary. Since structural relations are *unrelated* with one another, they can be organised in a list (see Figure 2.25).

![Structural Relations](image)

**Figure 2.25**
Structural Relations.

### Quantity Spaces

Quantity spaces represent an ordered set of values that a quantity may have. Quantity spaces have the following features that must be visualised: model-type, its unique identifier (class-name), the value types (*points* and *intervals*), the values' given names and the order between the values. As before, model-types are represented by icons and given names are represented by text labels. In this case the model-type identification for values
is necessary because they play a role in other contexts and therefore the model-type identification is essential. Characteristic icons can be used to visualise the nature of the values. Typically, a point is a single value and thus can be better illustrated by a point. Intervals, on the other hand, abstract from a set of quantitative values and are better represented by a line.

Different possibilities for visualising quantity spaces are suggested in Figure 2.26. In Figure 2.26a, the quantity spaces and the complete set of values associated with them are displayed simultaneously in one single area, the values being linked to the quantity space by connecting lines. A problem with this approach is that if the lines can move, the correct order of the values may get mixed up. If the lines are fixed we still have the problem of getting the screen cluttered when the number of quantity spaces increases.

In the second and third example, quantity spaces are arranged in a two column table. In these tables, the first column features a list of available quantity spaces. In the second column the set of values pertaining to the currently selected quantity space are displayed. The difference between the second and third example is the visualisation of the set of values. Should it be vertically or horizontally laid out? One may argue that displaying it horizontally is more intuitive because of the mathematical metaphor. Consider, for instance, having the value zero of a quantity space as the origin of the x-axis in the common Cartesian frame. All values placed on the right side are, obviously, bigger than zero and the ones that are placed on the left are smaller than zero. Contrarily, others may argue that presenting a quantity space vertically is more suggestive because it is more closely related to the physical world. For instance, a thermometer and a column height measure on containers. Moreover, vertical layout is not incompatible with the math metaphor when thinking of on y-axis.

Quantity spaces are best represented as suggested in either Figure 2.26b or 2.26c, with a slight preference for 2.26b. If a certain model ingredient is unrelated to others of the same type, it should be organised as a list. Additionally, by using a list the fixed order of the values is maintained.
Quantities

Quantities are the dynamic properties of entities. Again, an icon will be used to identify the model-type and a text label to display the class-name given by the model builder. Figure 2.27 illustrates the visual approach. Quantities are organised in a list due to the fact that they do not relate to each other. It is preferable to keep the model-type icon with the text label for consistency.

<table>
<thead>
<tr>
<th>Quantities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pressure</td>
</tr>
<tr>
<td>Temperature</td>
</tr>
<tr>
<td>Volume</td>
</tr>
</tbody>
</table>

Figure 2.27
Quantities View.

Model Fragments

*Model Fragments* model knowledge about the physical world. They contain a set of conditions that must hold for the model fragment to become active, in which case a set of consequences is added to the model. Conditions and consequences are stated in terms of the basic building blocks described earlier. The visualisation of a model fragment therefore consists of visualising all previously presented building blocks as well as the relations between them. In fact, model fragments are *structures*, which means that all knowledge elements contained in them are related to each other. This gives us a starting point of how model fragments should be visualised and organised.

The individual model ingredients constituting a model fragment, i.e. entities, attributes, quantities and quantity spaces were all previously described. For the purpose of visualising a model fragment it is relevant to note the implicit 'belongs-to' relationships between these model ingredients. Although, these implicit relations do not comprise a specific model ingredient type they have to be communicated to the model builder. *Quantities and attributes*, being properties of one and only one entity, should be explicitly grouped or connected in a similar way to the entity they belong to. *Quantity spaces* and *derivatives*, on the other hand, represent features of quantities and must be linked to the quantities they belong to. Figure 2.28 shows two possible ways of visually grouping or connecting this knowledge. In the first example of Figure 2.28 a visual container holds a series of slots to represent the entity's name and its features (attributes and quantities). In the second example of Figure 2.28 a different approach is taken in visualising the different knowledge components. In this case, every single model ingredient is represented by a node and the 'implicit' relation *belongs to* is represented by a line between two nodes. The fundamental difference between the two examples is that in the first case, all knowledge pertaining to an entity is grouped and laid out in a tabular form, while in the second
case it is represented in the form of a graph. In the former case the entity and all the knowledge associated to it forms a kind of 'atomic super node'. This form of layout is centered around the entity. In the latter case, all model ingredients are treated equally and may be freely organised by the model builder.

![Diagram](image-url)

**Figure 2.28**
Visualising building blocks in a model fragment.

Other knowledge types that may exist and need to be visualised in a model fragment are structural relations and dependencies. These knowledge types share the common characteristic of always relating two model ingredients. Structural relations are defined between two entities, while dependencies relate quantities, values, derivatives and combination of these. Figure 2.29 extends the examples of Figure 2.28 by including the visual representation for this type of knowledge. In both cases, relationships are represented by arrows. A structural relation is tagged by a text label with its name. Notice that, no icon representing the (structural relation) type is used. A structural relation is the only relationship type that can exist between two entities and, therefore, the given name is enough to uniquely identify it. In the case of dependencies, a pre-defined fixed set of possible dependencies exists and therefore the arrows representing them are tagged by characteristic icons suggesting their underlying model-type. For example, the \( > \) sign for an inequality, or an P- sign for a negative proportionality, both shown in Figure 2.29.

A problem with the approach illustrated in Figure 2.29a, the container approach, is that dependencies between quantities belonging to the same instance are somewhat difficult to represent, because they are shown as lines 'from' and 'to' the same super node. Particularly in the case of 'many' such lines the layout may become cluttered. Another problem arises in the case of inequalities between two quantity values, where we need to explicitly visualise between which specific values in the quantity spaces the inequality exists. The proposed solution in this case would be to label the start and the end of the arrow with the specific value names, see Figure 2.30. Again, this might be a potential cause for visual cluttering.

In the second approach illustrated in Figure 2.31 (the autonomous node approach), the model builder is granted more flexibility in laying out the model components. As such, dependencies between different quantities are represented by arrows starting and ending at different nodes. The nodes may be organised freely in order to increase the readability
2.3. DESIGNING MODELLING TOOLS

Figure 2.29
Visualising dependencies in a model fragment.

Figure 2.30
Distinct visual containers for Conditions and Consequences.
of the model. Also, inequalities between specific quantity values may be visualised by displaying the involved values under the quantity space, which can be moved around freely. The problem here, is that more space is claimed by the links between nodes, while these are implicitly in the tabular form. Also the information is less compact, yielding a less sparse visual presentation. While the tabular form is more coherent and applies well for complex models with many entities and not excessively many auto-dependencies, the autonomous node form is probably more adequate for simpler models or in the cases in which many auto-dependencies need to be specified.

Another issue concerns the visualisation of the 'conditions' and 'consequences' of a model fragment. Two approaches can be pointed out in this respect. The first approach, following the 'visual container' idea, uses separate components to distinguish between conditions and consequences, see Figure 2.30. This approach conveys a clear separation of the two blocks. However, model ingredients present in the conditions of a model fragment may need to be referred to in the consequences of the same model fragment, for example, further constraining a relationship between two quantities of two entities. In order to avoid connections crossing the border between the two blocks, the only feasible solution to deal with the above-mentioned situation, is to show the super node representing the entity twice, once in each block. That is the reason why the entity instances in the consequences block of Figure 2.30 are grayed out. However, from this visualisation it is not immediately obvious that both icons refer to the same instance. The alternative approach, shown in Figure 2.31, does not have two separate containers, but visually distinguishes by means of colour whether a component plays a role in either the conditions or the consequences. A reference copy of the model ingredient is then no longer needed. However, the model builder must be aware of the meaning of the two colours. For instance, a dependency added as a consequence would be coloured blue, whereas all the ingredients that are conditional would be coloured red.

In summary, the first approach spatially separates conditions and consequences and therefore renders the separation of roles more obvious. However, reoccurring ingredients in consequences may lead to confusion. The latter approach, on the other hand, is more compact.
2.3.3 Towards Interaction and Support

The previous Section discusses how the model ingredients that are used to build a qualitative model can be visualized on the screen. Seven workspaces have been put forward, five dealing with building blocks and two dealing with constructs. This section discusses the issue of how a model builder will interact with these workspaces in order to build a model. The overall design of the graphical user interface (GUI) to facilitate this interaction consists of the following aspects: builders, graphs, vertices, arcs and tools. Figure 2.32 illustrates these concepts and their relationships. In short, the application manages one or more builders and serves as the starting point for a GUI navigation. Every builder contains a series of graphs and other components, such as labels. A graph is a collection of vertices connected by arcs. These are placed and manipulated by the modeller using a mouse-controlled tool. Every tool has associated with it an interaction dialog. The interaction dialogs are secondary windows that allow the modeller to further refine the interaction with the builders.

Concerning the window management two approaches can be considered. Firstly, as used in many software systems [84, 10, 116], every builder could be placed in a different tabbed window with a fixed size. At a certain moment just one builder would be visible and the navigation through the builders could be easily done by selecting the right tab. Thus, tabbed windows avoid clutter up the desktop. The second approach is the dominant model for window management in desktop interfaces. It uses independent overlapping windows for the builders [8, 59, 29]. In this way, the modeller is responsible for resizing, laying out, moving, opening and closing the builder any time the modeller wants. Figure 2.33 illustrates these approaches. The drawbacks are the amount of time that has to be spent on arranging the screen into a suitable form as well as switching between builders. Despite the drawbacks of the independent overlapping builders, we believe that it is still more appropriate for supporting our task. Having multiple builders open will be essential, because next to supporting a model building step, these builders also provide the model builder with overviews of what has been built so far. In what follows, the details of designing builders, graphs, vertices, arcs and tools are given.
BUILDERS

Builders are interactive windows that support the modeller in building specific model parts. Every builder manages a collection of graphs and its layout depends on the specific goal of the builder. Each builder has associated with it context-specific tools, which are placed on a toolbar and/or a menu. As the focus changes from one builder to another, the available set of tools changes as well. There are some tools which are common to all builders and work in a similar way in every one of them. Other tools are specific to a given builder and therefore will only be available for that builder.

The GUI must provide mouse- and keyboard-based navigation within the builders. In order to promote usability and reduce the GUI’s learning curve, all builders must have a common look and navigation (e.g. the tools should always be at the same place within the builders. The way of accessing them should also be identical).

Graphs are graphical representations of domain concepts. Every graph contains vertices, which may be connected by arcs. With respect to the visualisation of the model ingredients, each vertex represents an individual model ingredient and an arc represents a model ingredient that expresses a relationship between two individual model ingredients. The goal of the graphs in the builders is to hold the visualisation of a structure that may consist of many inter-related model ingredients. For example, the Entity Builder consists of a graph, entities are the vertices and the lines between the entities, representing the hierarchy relation, are arcs. In our visualisation, every vertex may contain several graphic components: icons for representing the model-type of the ingredient and text labels for the given class-name (e.g. in the entity case, an icon for the type and a text label for the modeller-given class name). Also, every arc may contain an icon or a text label (e.g., an arc that represents an inequality will have an icon, and an arc for a structural relation will have a text label (model builder given name), see Section 2.3.1, and an arrow tip at each end (in order to specify direction).
2.3. DESIGNING MODELLING TOOLS

Tools

Tools are the means of interaction with a specific builder and with them, the modeller creates, modifies and organises the model ingredients within each builder. Every tool has an associated interaction dialog with common look-and-feel. In the following a description of the tools is given.

**PlaceVertex Tool**  Associated with a specific dialog and in a specific builder, the modeller can create the various types of vertices using the 'PlaceVertex Tool'.

**ConnectVertices Tool**  This tool in association with specific dialogs can be used to create arcs connecting vertices in graphs.

**Pointer Tool**  This tool is the default tool and is used to select and move around vertices and arcs in order to organise them visually on the screen. When one of these graphical primitives is selected, their handles should be made visible to the modeller.

**Edit Tool**  This tool is used to modify the properties of existing graphical objects (e.g. rename an entity or change its super type). By selecting an object and then clicking on the edit tool, the corresponding dialog should be displayed with the fields filled in with the object's actual properties.

**Delete Tool**  This tool is used to delete any object in the builders.

The basic functionality of a tool should always be the same (e.g. a ConnectVertices Tool will always be used to connect vertices), but variations specific to the task being performed in an active builder are introduced by using different dialogs. These dialogs have associated with them specific methods which are triggered by the modeller's interaction with the dialog controls. Thus, different functionality of the same tool is embedded in the implementation of the associated dialog and its methods. Table 2.1 shows the tools and dialogs associated for specific builders.

Basic Support

This section defines support as the design of interactive tools and controls that allow the modeller to manipulate the graph. The tools are, as previously described, builder dependent and context-sensitive. Being context-sensitive means that a tool is activated to operate on a particular graphical object only if this tool can assist the modeller in performing some task with that graphical object. Furthermore, context-sensitive features make the navigation options more visible and thus supporting exploration of the model builder interface. When an object is selected the modeller can see all available options for that object enabled within the tool set. For example, in a model fragment builder, when an entity graphical object is selected, the availability of the tools to add an attribute and to add a quantity should be made visible to the model builder.

When the basic requirements are fulfilled, and a tool can be used, the input-output dependencies within the subtask supported by that tool, can be used to determine whether
the modeller has performed the task correctly or at least sufficiently (i.e., syntactically speaking). For instance, within the 'PlaceVertex tool' when adding a quantity in a model fragment, the modeller always has to select the entity to which the new quantity must be applied. The task is not sufficiently completed without that information and thus finishing the task should be made impossible (of course, it can be cancelled). For each builder, the minimum required steps to interact with a tool have been identified. They should guide the design of the tools to support the modeller in always performing the task to a sufficiently complete level.

Table 2.2 summarises the minimum requirements to use the tools within each of the builders.

### Verification for Editing and Deleting

Two basic functionalities are editing and deleting model ingredients from the model. For example, selecting an entity in the Entity Hierarchy Builder, and then selecting the 'Edit Tool' will invoke a dialog with which the entity can be renamed and/or its super-type can be changed. Alternatively, the model builder may want to delete an entity from the hierarchy that is no longer necessary in the model. However, since the model ingredients are most of the time highly interrelated, the issue of handling model changes must also be investigated. Several constraints concerning editing and deleting a model ingredient can be formulated in order to guarantee the consistency of the model.

**Editing Model Ingredients**

Editing a model ingredient can only be done in the builder
where the model ingredient has been created. A common 'editing' task for all model ingredients is renaming them. In this case, if the model ingredient is used somewhere in the model, it must be updated. Certainly, the new given name must be a unique name in order to be valid.

Deleting Model Ingredients A model ingredient can only be deleted if it is not used anywhere in the model. For example, if an entity is part of a model fragment, it is mandatory to delete the occurrence of the entity in the model fragment before the entity can be deleted from the model in the Entity Hierarchy builder.

In short, when a model ingredient has a relationship with another model ingredient, it is necessary to delete first the relationship in order to be able to delete the model ingredient itself. In this way, the model builder is forced to be aware of all changes in the model being created.

2.4 Summary

This chapter has presented a discussion on designing a model building environment. The main purpose was to explore the tasks in modelling systems and their behaviour using a qualitative vocabulary, to define means for visualising the different model ingredients, and
also to specify ways for interacting with the model ingredients. The important conclusions of this chapter are as follows:

- Taking a rational task analysis as starting point in the design cycle is a way to clarify the tasks that must be performed when building qualitative simulation models. The results of the analysis consist of a set of manageable and distinct subtasks that a modeller is supposed to perform and which must be supported in a modelling environment. The results from the task analysis also suggest seven main separate modelling activities. They are the creation of entities, attributes, structural relations, quantity spaces, quantities, scenarios and model fragments.

- An important issue in design is how to visualise the information on the screen. This leads to the definition of general principles for graphically visualising the model ingredients. First, we should use icons for representing the model-type that is defined by the modelling language and text labels for the information provided by a modeller, class-name and instance names. Second, the visual representation for model ingredients that convey relations between other model ingredients can be made of a line with an icon or a text label connecting the related items. An icon must be used when the relation to be visualised involves one of the pre-defined dependencies provide by modelling language, for example an influence. The text label must be used in the case of structural relations. They are created by the modeller to represent relations between entities and, therefore, the modeller given name must be used as the identifier of those relations. Third, in order to preserve consistency, the icon that represents the model-type must always be used whenever a class of that model-type is displayed. Finally, relations that do not exist explicitly in the model building language can be represented as thin lines between the model ingredients (e.g., the 'implicit' relation 'belongs-to' between an entity and a quantity).

- Based on the criteria for visualising model ingredients, alternative visualisations and organisations of these model ingredients on the screen were presented. Two main differences between alternative visualisations within model constructs are discussed. In one approach, we have the entity and all the knowledge associated to it (attributes and quantities) forming an 'atomic super node'. This form of layout is centered around the entity. In the other approach, all model ingredients are treated equally and, therefore, each of them is an individual node and may be freely moved and organised by the modeller. Both approaches have advantages but, the latter approach seems to be more efficient because relations between the various model ingredients exist and they can be made explicit by an arc from and to individual nodes. The second main difference between the presented alternatives concerns the conditions and consequences in a model fragment. In one approach we have two distinct areas for representing them and, therefore, it is immediately clear what belongs to one or to the other part. The other approach uses colour coding to make this distinction, and avoids the drawback of having a copy of a model ingredient in the consequences when it plays a role in both sides. Therefore, this approach is preferred.
2.4. SUMMARY

- The conceptual design for a model building environment includes the concepts of builders, tools, graph, vertices and arcs. The builders are the workspaces for building specific model ingredients. They consist of graphs that consist of vertices (model ingredients) and arcs (model ingredients that relate two other model ingredients). In order to interact with the builders we have tools which are used to create ('PlaceVertex Tool', 'ConnectVertices Tool'), modify ('Edit Tool'), and delete ('Delete Tool') model ingredients within the builders. Also, vertices and arcs can be moved and selected in the builders using a default tool ('Pointer tool'). Next to this basic functionality each builder has dedicated functionality. The behaviour and functionality of a tool is driven by the specific interaction dialog associated with it in a specific builder.