

Tools for knowledge articulation software in education

Maarten van Hoof

Master thesis at the University of Amsterdam
Department of Social Science Informatics
Roetersstraat 15, 1018 WB Amsterdam, The Netherlands

Amsterdam, June 2003

Abstract

Knowledge articulation software enables learners to create models and by doing so acquire insights into the behavior of systems. Tools that bridge the gap between sophisticated knowledge representation languages and modern application development techniques are needed in order to facilitate the creation of such model-building software. This document describes the design and implementation of such a tool, the GKOM module, which serves as a basis for knowledge articulation applications. GKOM offers application developers a knowledge representation language based on qualitative reasoning, and it provides modeling support. Additionally, GKOM-based applications operate in a network for collaborative modeling and learning, in which communication among modeling applications facilitates collaborative learning, and communication with remote services allows for advanced features such as simulation, and central storage of models.

Supervised by Dr. Bert Bredeweg

Contents

1	Introduction.....	5
2	Qualitative reasoning in education.....	7
2.1	Knowledge articulation software.....	7
2.1.1	Qualitative Reasoning.....	8
2.1.2	Qualitative Reasoning in education.....	9
2.2	Facilitating the use of QR in educational software.....	10
2.2.1	Facilitating a knowledge articulation application.....	11
2.2.2	Facilitating a community of practice.....	12
2.3	Conclusion.....	13
3	Qualitative reasoning and GARP.....	15
3.1	Qualitative reasoning with GARP.....	15
3.2	The world according to GARP.....	16
3.3	An example model: the u-tube.....	21
3.4	Conclusion.....	24
4	GKOM design.....	25
4.1	GKOM and the modeling environment.....	25
4.2	Sub-modules of the GKOM module.....	25
4.3	Design of the knowledge representation.....	27
4.3.1	A very short introduction to Object Oriented Programming.....	27
4.3.2	Three knowledge levels, ten modeling primitives.....	28
4.3.3	The role of the entity hierarchy.....	32
4.3.4	Model, simulation, state, and transition.....	32
4.4	Conclusion.....	33
5	Offering support.....	35
5.1	Introduction.....	35
5.2	Creating knowledge.....	35
5.2.1	Creating elements at the type-level.....	36
5.2.2	Creating elements at the occurrence-level.....	37
5.3	Modifying knowledge.....	38
5.3.1	Changing the structure of the entity hierarchy.....	39
5.3.2	Modifying quantity spaces at the type-level.....	42
5.3.3	Modifying scenarios and model fragments.....	44
5.4	Removing elements from a model.....	45
5.5	Implementing support.....	46
5.6	Concluding remarks.....	48
6	GKOM fundamentals.....	49
6.1	Overview.....	49
6.2	Top-level classes.....	50
6.2.1	The KnowledgeObject class.....	50
6.2.2	The Type class.....	51
6.2.3	The Occurrence class.....	52
6.2.4	The Instance class.....	53
6.2.5	The Model class.....	54
6.2.6	Simulations.....	56
6.2.7	Overview of the top-level classes.....	60
7	Building blocks.....	63
7.1	Entities.....	63
7.1.1	EntityType.....	63
7.1.2	Entity.....	64
7.1.3	EntityInstance.....	65

7.2	Properties.....	65
7.2.1	PropertyType	65
7.2.2	Property	66
7.2.3	PropertyInstance	66
7.3	Relations.....	67
7.3.1	RelationType	67
7.3.2	Relation.....	67
7.3.3	RelationInstance	68
7.4	Quantity Spaces.....	68
7.5	Quantities	70
7.5.1	QuantityType	70
7.5.2	Quantity	70
7.5.3	QuantityInstance.....	71
7.6	Values.....	72
7.6.1	ValueType	72
7.6.2	Value.....	72
7.6.3	ValueInstance	73
7.7	Derivatives	73
7.7.1	DerivativeType	73
7.7.2	Derivative	74
7.7.3	DerivativeInstance	74
7.8	Dependencies	75
7.8.1	Dependency	76
7.8.2	DependencyInstance	76
7.9	Inequalities	77
7.9.1	Inequality.....	77
7.9.2	InequalityInstance.....	78
7.10	Correspondences	78
7.10.1	Correspondence	78
7.10.2	CorrespondenceInstance.....	79
7.11	CausalDependencies.....	79
7.11.1	CausalDependency	80
7.11.2	CausalDependencyInstance	80
8	Model fragments and scenarios.....	81
8.1	Scenarios	82
8.1.1	ScenarioType	82
8.1.2	ScenarioInstance.....	83
8.2	Model Fragments.....	84
8.2.1	ModelFragmentType	85
8.2.2	ModelFragment	87
8.2.3	ModelFragmentInstance	88
8.3	Conclusion.....	89
9	Conclusions and discussion	91
9.1	Conclusions	91
9.2	Discussion	92
9.3	Further research.....	93
10	Literature:.....	95

1 Introduction

A modern approach to education is learning by doing. This approach to learning has its roots in constructivism (Vygotsky, 1978), which holds that learners actively construct their knowledge rather than passively receive it from the environment. One form of learning by doing is learning by building models, in which learners formalize their understanding of a domain in a model (e.g. Forbus, 2001). The idea is that in this manner learners develop more fundamental insights. Moreover, the models they create facilitate knowledge communication between learners.

Researchers developing software tools to support this type of learning, build modeling applications that offer a *graphical knowledge representation language*. Such a language consists of a vocabulary of graphical symbols representing abstract concepts and a grammar that states how these symbols can be used to represent knowledge. A simple example of a graphical knowledge representation language is the *concept map* (Novak, 1977), a map of concepts related by relationships. But concept maps are very general: both the concepts and the relationships can be of any nature. More elaborate languages have a richer vocabulary.

One line of Artificial Intelligence (AI) research that has produced languages with a rich vocabulary is research on qualitative reasoning (De Kleer & Brown, 1984; Forbus, 1988; Bredeweg, 1992). Qualitative reasoning (QR) is particularly apt for modeling tools used in education because of its foundation in commonsense knowledge formalisms (Hayes, 1978) and its focus on causality. These features facilitate automatic generation of feedback and explanation (Falkenhainer & Forbus, 1991; Vadillo et al, 1998). Additionally, languages based on qualitative reasoning offer qualitative prediction of behavior, enabling the use of model simulations as a feedback mechanism (Sime & Leitch, 1992; Sime, 1998). But to apply these languages in educational modeling tools, two problems must be solved. First, the complex nature of these languages forces application builders to provide modeling support to learners that use these languages. Second, these languages are in the wrong form: they are not very compatible with modern application-development techniques. They are text-based rather than graphical, requiring that a graphical interface be built to operate on them. And they are often based on declarative programming languages rather than the procedural languages used to develop end-user applications. These two factors make application development a time-consuming task. Tools are required to overcome these problems and apply qualitative reasoning techniques in educational modeling applications.

This document describes the design and implementation of such a tool: GKOM. GKOM is a module that serves as a basis for a knowledge modeling application. It offers a knowledge representation language based on qualitative reasoning, a system for modeling support, and network capabilities for remote simulation. GKOM is written in Java, a modern, object-oriented programming language that is often used to build end-user applications. The objectives for building this module are twofold: it should facilitate the creation of modeling applications that use qualitative reasoning as a formalism for knowledge modeling, and it should place these applications in a framework for distributed modeling and learning.

This document is organized as follows. Chapter 2 explores the field of knowledge modeling tools used in education and proposes an architecture for collaborative modeling and learning. It introduces qualitative reasoning as a suitable knowledge representation language and looks at some of the current educational software based on qualitative reasoning. The last section of chapter 2 suggest the creation of a module to facilitate the creation of knowledge modeling software. This module, as a component in the architecture for collaborative modeling and learning, will handle knowledge capturing, modeling support and networking capabilities.

Chapter 3 introduces GARP as a tool for qualitative reasoning. It describes the GARP vocabulary and the concept of qualitative simulation. The final section presents a simple qualitative model that is used as an example throughout this document.

In chapter 4, the overall design of the module is described. It offers a detailed view of GKOM's place in the modeling process, and introduces the sub-modules that make up the GKOM module and handle knowledge capturing and support, interfacing with the modeling application and communication with remote applications. Chapter 4 also introduces the design of the GKOM knowledge representation.

Modeling is an error-prone process and modelers need guidance in their use of a knowledge representation language. Chapter 5 describes the nature of the modeling support built into the GKOM module.

Chapters 6, 7, and 8 are entirely devoted to the implementation of the GKOM knowledge representation. Chapter 6 describes the basics of the knowledge representation in terms of the first two levels of the class-hierarchy. Chapter 7 describes 'building blocks'; the model ingredients used to create the entity hierarchy and the model fragments and scenarios of a model. Chapter 8 describes the model fragments and scenarios themselves.

The concluding chapter, chapter 9, describes our experience with building the GKOM module. The conclusions reflect on the goals with which the module was built and to what extent they have been reached. The discussion looks at possible criticism of the module in terms of goals that have not been reached and in terms of side effects of the implementation. The final section suggests directions for further research.

Acknowledgements

I would like to thank Bert Bredeweg, my supervisor, for the time he spent on this project. Our frequent discussions in the development phase have improved GKOM and have encouraged me to implement features that I now consider indispensable. This document has benefited greatly from his constructive criticism.

I have my parents and my girlfriend to thank for emotional support and for bringing me back up to speed when things were moving slowly. I especially enjoyed meetings with my father who, not bothered by any knowledge about programming, qualitative reasoning or creating educational software, gave me valuable advice about writing this thesis.

Last but not least, I thank my employer for being flexible enough to allow me days off on often very short notice.

2 Qualitative reasoning in education

This chapter explores the current status of modeling applications used for educational purposes and describes how the creation of these applications can be facilitated. The first section introduces knowledge articulation software. It describes some of the current modeling applications used in education and underscores the importance of a proper knowledge articulation language. Section 2.1.1 suggests qualitative reasoning for this purpose, and section 2.1.2 describes how qualitative reasoning has been used in educational software.

In section 2.2 we present our ideas on how to facilitate the creation of knowledge articulation software. We suggest building an application-independent module that implements functionality required by modeling applications and look at it from two angles: as a stand-alone modeling application and as a component in a framework for collaborative modeling.

2.1 Knowledge articulation software

Much of the recent research on educational software focuses on modeling (*Quorum*, Cañas, 1995; *CyclePad*, Forbus et al, 1999; *STAR-light*, De Koning et al, 2000; *Betty's Brain*, Leelawong, 2001). Applications created in this field typically require learners to formulate their understanding of a domain in a graphical knowledge representation language. We will use the term *knowledge articulation software* to describe these applications. This approach to learning has its roots in constructivism (Vygotsky, 1978), which holds that learners actively construct their knowledge rather than passively receive it from the environment. Modeling applications allow learners to literally construct their own representation of the world in the form of a model. Modelers articulate relationships between concepts and dependencies between their beliefs, and develop a more profound understanding of the domain in the modeling process. At the same time, models provide a means to externalize thought, allowing learners to work through more complex problems. Graphical knowledge representations make knowledge insightful, and increase knowledge communicability. This helps learners present their ideas to others for discussion and collaboration (Forbus et al, 2001).

A *knowledge representation language* is a set of abstractions and conceptualizations about how to formulate knowledge; it is a *language* for capturing knowledge. Knowledge representation languages typically offer a vocabulary of abstract terms such as *class*, *object*, *relation*, and *attribute*; terms that are general enough to describe a broad range of domains. The grammar of a knowledge representation language prescribes how the language is to be used and could specify such things as that *objects are instances of classes* or *a relation relates two objects*. A *model* is knowledge represented in a knowledge representation language. Modeling applications for education offer a *graphical knowledge representation language*, knowledge representation languages for which graphical representations exist for the language vocabulary. It is the graphic nature of these languages that makes them suitable for education: graphical representations make the knowledge more insightful (Kulpa, 1994).

Modeling applications differ in the richness of their knowledge representation language. An example of an application that uses a relatively limited vocabulary is *Quorum* (Cañas, 1995), which allows modelers to create concept maps (Novak, 1977). A concept map is a set of concepts (objects, ideas, processes, etc) with relationships of any nature between them. Other applications have more elaborate vocabularies that require learners to differentiate between objects, attributes and variables, for example. Some applications feature what Forbus (2001) refers to as a 'construction kit': a set of generic objects, attributes and variables that learners use as building blocks for their models.

Modeling has some clear benefits to the learner, as explained above, but what makes modeling specifically suited for educational *software* is that formal representations of knowledge allow computers to reason with what learners articulate. In fact, a lot of the knowledge representation languages found in modeling applications have their origins in artificial intelligence research on problem solving (e.g. Russel & Norvig, 1995). The modeling applications are graphical user interfaces to languages that were once entirely text-based. The reasoning abilities of these languages are used to simulate the models that learners create, providing feedback about the correctness of the model.

Exactly how to represent knowledge is an important question. Knowledge representation has been an area of research in artificial intelligence for as long as the field exists, but those used in educational settings have different requirements than many of the traditional representation languages, which focuses on problem solving. A language used in educational modeling software must be both intuitive and expressive. A knowledge representation language that is often used in knowledge articulation software is qualitative reasoning.

2.1.1 Qualitative Reasoning

Qualitative reasoning (De Kleer & Brown, 1984; Forbus, 1988; De Kleer, 1990; Bredeweg, 1992) is concerned with reasoning about the behavior of systems in qualitative terms. The systems under scrutiny are usually physical systems, although models of systems in different domains have also been created (see Salles & Bredeweg (1997) for an example in ecology). The behavioral aspect studied most is qualitative prediction of behavior: the analysis of how the behavior of a system changes over time, based on a qualitative description of the system.

One of the early sources of inspiration for the field of qualitative reasoning is the work of Patrick Hayes (1978). Hayes introduced the term ‘naïve physics’ to describe commonsense knowledge that people have of the everyday physical world: about objects, shape, space, movement, substances, time, etc. While it is this sort of knowledge that allows people to function in the physical world, no formalization of this type of knowledge exists. In fact, physics laws are all based on the presupposition of a shared unstated commonsense prephysics knowledge. With the advent of Artificial Intelligence and its focus on autonomous agents that must function in the physical world, it became necessary to formalize commonsense knowledge in a way that was understandable for computers. Hayes felt that in doing so, AI overemphasized on toy worlds: overly simple models of the physical world that would only work in toy domains. He held that focusing on such small domains would never lead to an adequate formalization of commonsense knowledge and that without such a large-scale formalization, AI would never find out what the real problems of knowledge representation are.

An interesting note about Hayes’ work is that he very explicitly did not propose to create a computer program that would be able to use the naïve physics theory in some way. In fact, he felt that this would divert the attention from the main goal (building the naïve physics theory) and can be dangerous. It too is easy conclude that, because one has a program that works, its representation of knowledge must therefore be correct. Sometimes, representational devices turn out to be traps as people try to overcome difficulties generated by the representation itself. According to Hayes, the focus should be on content, not form.

The field of qualitative reasoning grew out of individuals who were inspired by the idea of a formalization of commonsense knowledge, but did not take Hayes’ advice about staying away from implementation or representation. An important idea behind qualitative reasoning is that it offers a more natural way of representing knowledge. Humans solve physics problems by first making a qualitative analysis of the situation. They use qualitative terms like hot and cold, or high and low, or increasing and decreasing and their qualitative analysis includes concepts such as one quantity having a positive influence on another quantity, or a particular value of one quantity leading to an increase in another quantity. Equations to calculate the exact answer are used *after* the qualitative analyses, but only if the question requires a quantitative answer. Many physics problems can be solved in all-qualitative terms. A qualitative analysis is crucial for comprehending the problem and writing down the appropriate equations. People do not believe the answers predicted by equations unless these answers can be supported by an intuitive understanding (De Kleer & Brown, 1984).

The basic concepts of qualitative knowledge representations are:

- **Classes, objects and structure.** Structure is described in terms of objects, relationships between objects and attributes of objects. Structures make it possible to reason about behavior emerging from structure. Objects belong to classes and classes are embedded in a hierarchy.
- **Quantities and quantity spaces.** Behavior is described in terms of values and derivatives of quantities. Quantities have a small set of possible values, captured in a quantity space.

- **Quantity relations.** Causality is described using relations between quantities, values and derivatives, which make it possible to express such things as that one quantity having a particular value causes another quantity's value to increase.

Many of the existing tools for qualitative analysis also support qualitative simulation. Qualitative simulation is the prediction of behavior of a system, based on a qualitative analysis of that system. The result of a qualitative simulation typically consists of a set of states and state transitions. Qualitative simulation is discussed in chapter 3.

2.1.2 Qualitative Reasoning in education

Qualitative reasoning focuses on causal accounts of physical mechanisms that are easy to understand, and allows reasoning in the absence of specific numerical data. Its focus on concepts and causality makes it particularly apt for communication and explanation in educational settings (Bredeweg en Winkels, 1998). Forbus (2001) identifies two reasons why qualitative reasoning¹ is particularly appropriate for application to science and engineering education. The first is that *qualitative reasoning represents the right kind of knowledge*. Much of what is taught in science in elementary, middle, and high school consists of causal theories of physical phenomena. Traditional mathematical and computer modeling languages do not attempt to formalize such knowledge. In qualitative reasoning, uncovering how people think about physical entities and processes is one of the central issues, and progress in qualitative reasoning has led to new modeling languages that describe entities and processes in conceptual terms, embody natural notions of causality, and express knowledge of the modeling process itself.

The second reason that qualitative reasoning is particularly apt for science and engineering applications is that *qualitative reasoning represents the right level of knowledge*. Principles governing a domain need to be mastered at the qualitative level to provide the kind of deep, robust understanding that engineering education seeks to impart.

Qualitative reasoning has been used in educational software in a variety of ways:

- To generate explanations (Falkenhainer & Forbus, 1991; Vadillo et al, 1998). Qualitative reasoning focuses on causal accounts and concepts, and these provide a solid foundation for generating explanations in natural language. One of the primary reasons for using qualitative reasoning in education is the belief that it resembles human reasoning, and it should thus be relatively easy to generate 'natural' explanations of phenomena.
- To generate questions and assignments. An analysis of the qualitative description of a system and the simulations based on that description can identify causal paths in a system's behavior. The processes and quantities involved in these paths can be made the subject of automatic generation of questions and assignment. Recent research in this area is described in Goddijn (2003).
- To analyze a situation in order to create a quantitative model. In CyclePad (Forbus et al, 1999), an *articulate virtual laboratory* for creating thermodynamics cycles, a qualitative analysis of the cycle created by the student is used to generate a quantitative model of the cycle. This model is subsequently used to solve mathematical questions concerning the cycle.
- To create interactive and articulate modeling and simulation environments. Qualitative reasoning terminology is intuitive enough for use in an educational setting, and simulations based on the models that learners create can be used as feedback (Sime & Leitch, 1992; Sime, 1998; Bouwer & Bredeweg, 2001).

This document focuses on the last of these uses: creating interactive and articulate modeling environments. Forbus et al (2001) note that many of the current modeling environments used in education neglect three key issues in understanding the art of modeling:

- The importance of broadly applicable principles and processes. Existing educational modeling systems treat each modeling task as a new problem, with no connections to other

¹ Forbus uses the term qualitative physics rather than qualitative reasoning.

situations. This misses opportunities to help students see that the same principles and processes operate across a broad range of situations.

- Understanding when a model is relevant. Existing educational modeling systems do not address the issue of when a model is applicable and thus do not help students connect their models to real-world concerns.
- Qualitative understanding of behavior. Traditional modeling systems tend to be numerical, but providing all the data needed to run a numerical model can distract students from understanding the causal phenomena in the situation. In addition, a student may not be able to easily interpret outcomes in mathematical terms.

Qualitative reasoning provides many of the pieces needed to address these problems. Enabling and encouraging students to create their own domain theories should help them understand the broad applicability of scientific principles and processes. Qualitative reasoning provides the expressive power needed to state modeling assumptions and reason about relevance. Qualitative reasoning allows students to formulate intuitive, causal models of a domain.

2.2 Facilitating the use of QR in educational software

The goal of this project is to facilitate the creation of knowledge articulation software. Most of the features discussed in this chapter (graphical representations of knowledge, modeling support, automatic generation of questions, assignments and explanation) require a substantial amount of work to implement. Our goal is to create a module in an architecture for collaborative modeling and learning. This module should offer:

- **Reusability.** Application builders must be able to use it as a basis for their application, adding specific features to the core functionality implemented by the module.
- **A knowledge representation language.** Creating the knowledge representation itself is one of the more time-consuming tasks in building knowledge articulation software, and thus one of the most valuable features to offer.
- **Modeling support.** Modeling is an error prone process and modelers need guidance in their use of the knowledge representation language.
- **Communication with remote services.** The architecture for collaborative modeling and learning will contain other components, such as simulation services or model repositories. The component should be able to communicate with those.
- **Communication with other modeling applications.** True collaborative modeling requires direct communication between modeling applications, allowing modelers to share fragments of models with each other.

We have two objectives in building this module:

- **Facilitate the creation of knowledge articulation software for use in education.** The first three points above are related to this objective. Application developers must be able to build their software on top of the functionality offered by our component, allowing them to spend more time on features such as user interface and user interaction or higher-level modeling support.
- **Facilitate the creation of software to support communities of practice in education.** The last two points above are related to this objective. The component should implement a protocol for communication with remote services and other modeling applications.

The next two sections describe the module, which we have called GKOM (GARP Knowledge Object Model), in terms of its place in the architecture for collaborative learning and modeling and in terms of its features.

2.2.1 Facilitating a knowledge articulation application

Figure 2-1 shows a knowledge articulation application based on GKOM. From a high-level perspective such an application consists of two modules: the GKOM module and a module built by the application developer. The developers' module is a graphical user interface to the functionality offered by the GKOM module and adds application-specific features.

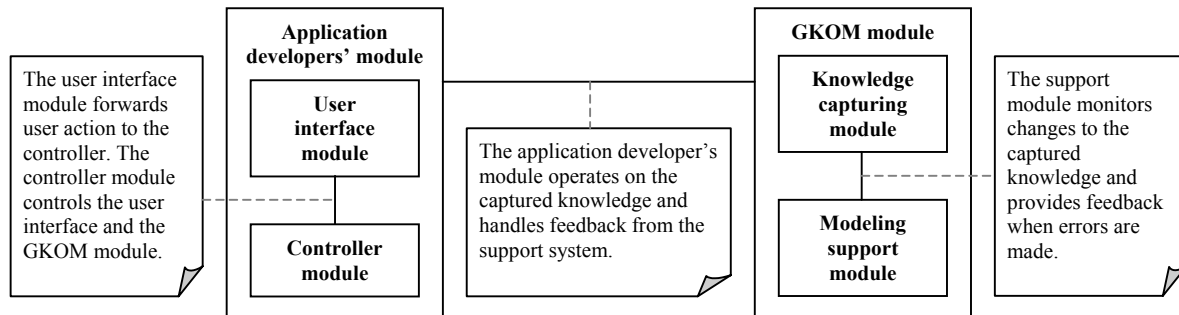


Figure 2-1: A knowledge articulation application based on GKOM.

The application developer's module

Although GKOM makes as little assumptions as possible about the inner workings of the application developers' module, such a module can be expected to at least consist of a user interface module and a program control module. The user interface module is the front-end to the application. The controller module controls the user interface and handles the commands that the user issues and the feedback it receives from the GKOM module. The user interface handles interaction and representation. It allows the user to interact with the application by offering a set of controls (buttons, menu's, dialogs, etc) that operate on the other components of the application. Interaction is concerned with questions such as 'how does the user create a new object?' or 'how does the user start a simulation?' Representation is concerned with offering a view on the current state of the application. Representation is concerned with questions like 'what does an object look like?' or 'how are the results of a simulation displayed?'

Creating an intuitive graphical user interface is the main reason for building a modeling environment in the first place. Qualitative models in their 'natural' form (usually a structured text file containing first-order logic statements) are too complicated to be understood by non-experts. A modeling environment should provide an intuitive visual language for representing knowledge, and an intuitive way for the user to interact with the model. This allows non-experts to use the expressive power of qualitative models, without requiring them to spend a lot of time learning about qualitative reasoning. This is easier said than done. Finding the appropriate visual representations for objects, relations, properties and quantities, or for abstract notions like conditions and consequences, is the subject of a sizeable portion of the research done in the field of modeling (e.g. Bessa Machado & Bredeweg, 2001). It is well beyond the scope of this document. In fact, it is the aim of this project to create a module that handles as much of the functionality required in a modeling application as possible, so that application developers can spend more time on representation and interaction issues.

A knowledge articulation language

Qualitative models are usually stated in a declarative language like Prolog. But a modeling application is usually written in a procedural language such as C or Java and requires a knowledge representation in its native language. This *internal representation* is in essence a mapping of qualitative concepts to the concepts of the programming language used in the application and should be designed for interoperability with the graphical user interface, the support system and other components of the modeling environment.

Designing an internal knowledge representation in Java was the original goal with which we started this project (hence the name GARP Knowledge Object Model), and the GKOM *knowledge representation* is an important part of the GKOM module. The knowledge representation is outlined in 4.3 and extensively described in chapters 6, 7, and 8.

A support system

Modelers make mistakes in their use of a modeling language and a support system should at least provide the modeling-equivalent of spelling and grammar checkers. But a support system for creating qualitative models can be much more comprehensive than that. For example, student models can be compared to normative models to generate suggestions for improving a model. Simulations can be used to actively determine the consequences of changes made to a model. Automatically generated explanation can clarify unexpected simulation results.

These more comprehensive forms of modeling support are research themes in their own right (e.g. Bouwer & Bredeweg, 2001). Offering them is beyond the scope of this project, but the modeling-equivalent of spelling and grammar checkers is not. The support offered by GKOM is detailed in chapter 5. Additionally, application builders can use other GKOM features to implement more comprehensive support. The import-export module will assist in reading normative models, for example. As described above, GKOM offers access to a simulation service.

2.2.2 Facilitating a community of practice

Figure 2-2 presents a high-level overview of a distributed modeling environment. In such an environment, knowledge articulation applications communicate with other applications and with modeling services on the network. Separating the services from the applications has two important benefits: the services can be managed at one single location and the applications can be kept relatively simple, because they can leave advanced functionality to the services.

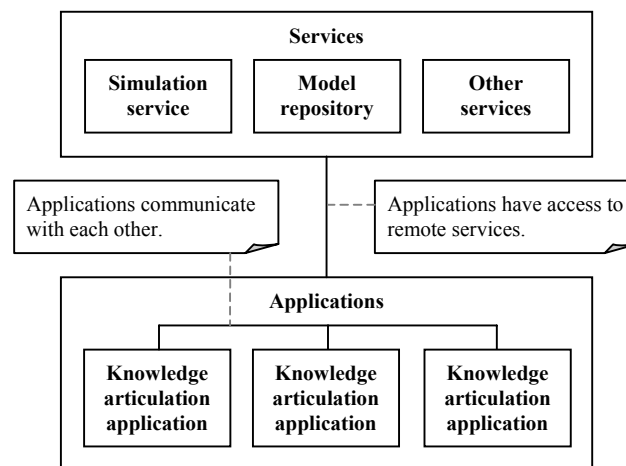


Figure 2-2: A distributed modeling environment.

To make a GKOM-based knowledge articulation application operate in an environment like the one depicted in Figure 2-2, the GKOM module includes a communication module.

A simulation service

Many of the advantages of using qualitative reasoning in educational software relate to the fact that it makes qualitative prediction of behavior by a qualitative simulator possible. The ideal modeling application includes a simulator, but in classroom situations, where computing resources are often scarce, including a simulator in the modeling application is not always the best solution. It is better to set up a remote *simulation service* and have the modeling application access that.

For qualitative reasoning with GARP, a simulation service already exists. The protocol for communicating with this service needs to be implemented by the communication module. This is detailed in chapter 4. Additionally, GKOM must provide a way to export models to a format that can be transmitted over a network connection, and to import models that originate from the network. These tasks are carried out by the import-export module introduced in chapter 4.

A model repository

A model repository is a central location for accessing and storing models. There are numerous reasons for creating a repository of models. First, one of the central ideas of qualitative modeling is that principles and processes found in one domain can be extended to other domains. A model repository can facilitate this by making all models available for future reference and use. Second, a model repository can serve as a portfolio of models made by one student or a group of students. This promotes reflection and captures how the student's understanding of the domain advances. It also allows for comparing models made by students in the classroom. Third, a model repository can achieve, or at least bring closer, the original goal of naïve physics as formulated by Hayes in his naïve physics manifesto (Hayes, 1978), namely the formalization of a sizeable portion of our everyday knowledge.

In its simplest form, a model repository may be a directory on a file server where models are stored. This requires a knowledge representation that can be saved as a file, a function offered by the import-export module. A more comprehensive solution would be a remote *repository service*: a machine on a network that allows modeling applications to store and retrieve models. This would be relatively easy to implement for an application builder, because GKOM already has built in support for accessing a simulation service. Accessing a repository service only requires minor changes.

Direct communication between applications

Collaboration is a valuable learning-technique and visual representation languages are excellent vehicles for communication about knowledge. A modeling environment in education can facilitate collaboration by allowing students to send fragments of their models to others, so that student can compare each other's models. Ideally, students can reuse parts of each other's models in their own models.

In its present implementation, GKOM does not offer support for collaboration. However, any implementation that an application builder would create will make use of the import-export module and the communication module.

2.3 Conclusion

This chapter has introduced qualitative reasoning and qualitative reasoning in education, and has analyzed the possibilities for facilitating the development of QR-based educational software. This analysis resulted in a set of functions that the GKOM module must offer. Chapter 4 continues on that subject and describes the design of the GKOM module. Readers who are familiar with GARP can skip ahead to that chapter. For those who are not, the chapter 3 introduces the qualitative reasoning tool that GKOM interoperates with: GARP.

3 Qualitative reasoning and GARP

GKOM was designed for interoperability with a qualitative reasoning engine called GARP (General Architecture for Reasoning about Physics, Bredeweg, 1992). GARP is an implementation of the QR principles described in the previous chapter and serves as a tool for qualitative analyses and qualitative simulation. The interoperability between GKOM and GARP is discussed in chapter 4, this chapter describes GARP itself. In section 3.1, the functioning of GARP is outlined in general terms. Section 3.2 describes the GARP language in more detail. Section 3.3 presents an example GARP-model.

3.1 Qualitative reasoning with GARP

GARP is a tool for qualitative analyses and simulation. It is given a description of a system in qualitative terms, and generates a set of possible states of behavior of that system. This section aims to give a description of GARP that is specific enough in the context of this document, but it is by no means complete. For other descriptions of GARP see Jellema (2000) or Bredeweg (1992).

At a high level, we can distinguish four concepts in a GARP knowledge representation:

- **Building blocks** are objects, attributes and quantities of objects, values and derivatives of quantities and dependencies between quantities, values and derivatives. They represent things in the real world and are the ‘stuff’ that scenarios, model fragments and rules are made of.
- **Scenarios** are aggregates of building blocks that describe a system in its initial state. They are the starting point of a simulation.
- **Model fragments** describe small pieces of domain knowledge in terms of conditions and consequences. A model fragment could specify, for example, that if an object *liquid* exists, then *liquid* will have a quantity *amount* and the value of amount will be greater than *zero*.
- **Rules** describe how the simulated system changes over time. An example rule could state that if a quantity is at a certain value and increasing, it will be at a higher value in the next state of behavior.

Given a scenario, a library of model fragments and a library of rules, GARP simulates the possible behavior of a system, starting at the initial state described in the scenario and stopping when it cannot infer any more new states of behavior.

Figure 3-1 illustrates the simulation process. The process starts with a scenario; an initial state. GARP then scans the library of model fragments to find those that apply to the scenario. It adds the consequences of the matching model fragments to the initial state and subsequently starts looking for rules that apply to the current description. This step yields a number of states. Then the process repeats: the new states form the input to another search for matching model fragments. The loop is traversed until it no longer produces new information.

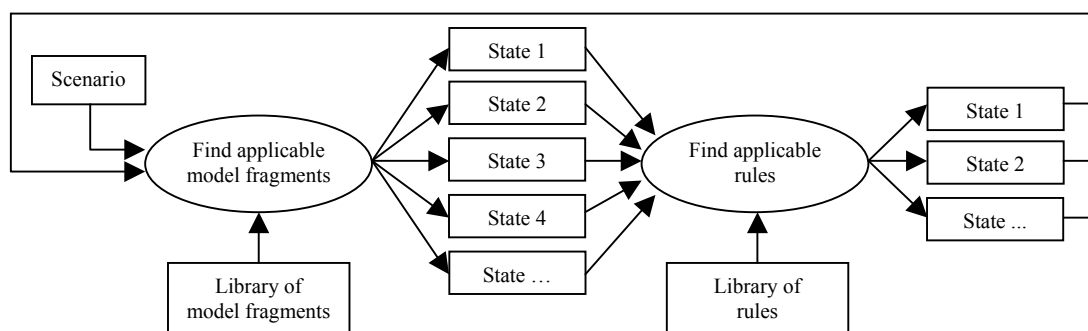


Figure 3-1: Inferring states of behavior

The task of a GARP user building a model of some real-world system is to create the library of model fragments and the library of rules, and the scenario that will serve as the initial state. More often than not, pieces of other models can be reused. The library of rules, for example, contains a set of basic rules that are used in every model. The rule specifying that an increasing quantity will reach its next value is a good example of a rule that is general enough to be used in any domain.

Model fragments are not usually that general, but many do exist that are general enough to be applicable in at least a set of related models. An example is the model fragment given above, stating that if a liquid exists, it must have a quantity amount of at least greater than zero. This model fragment can be used in any model involving liquids. A more elaborate example is a model fragment describing the flow of heat: heat flows from objects with a high temperature to objects with a low temperature. This applies to a cup of coffee cooling down because heat flows from the coffee to the surrounding air, it applies to water in a boiler heating up because heat flows from a heater underneath the boiler to the water inside the boiler, and it applies to the air inside a refrigerator cooling down because heat flows from the air to the cooling-liquid. A model fragment describing a heat flow can be used in any of these domains. This is what Forbus refers to when he states that using QR in education can underscore the importance of broadly applicable principles and processes (Forbus et al, 2001).

Reuse has its limits, however, and a modeler inevitably has to create model fragments that are specific to the real-world system that is being modeled. The modeler may also choose to add any number of domain-specific rules, although in practice this is less common.

3.2 The world according to GARP

This section deals with the knowledge representation used by GARP. A knowledge representation is a set of abstractions and conceptualizations about how knowledge can be formulated. It is a *language* used to talk about knowledge and must not be confused with the knowledge itself. Unfortunately, many of the terms used to describe a knowledge representation apply to both the representation language and the represented knowledge. The words *model* and *ontology* are examples of terms that are often used to describe both. To avoid confusion, this document uses the terms *knowledge representation* or *formalism* to describe the language in which knowledge is formulated, and the term *model* to describe formulated knowledge. The term *ontology* is used later in this document to describe a subset of the knowledge in a model. The following sections describe the concepts of the knowledge representation used by GARP.

Entities

Entity is the term for what has been called *object* until now. An entity represents a ‘thing’ in the real-world system being modeled. Entities can be defined generically or as instances. On a generic level, entities are defined in the is-a hierarchy, as *concepts* or *generic types*. The is-a hierarchy shows how entities inherit from one another; it is a tree-like structure that allows a modeler to specify that a *liquid*, for example, is a specific type of *substance*, or that *water* is a specific kind of *liquid*.

When used inside model fragments, scenarios, or rules, entities are called *instances*. An instance does not represent a generic object or a class of objects; it represents an actual object of a particular class. A modeler may make a model that has an entity *container* in the is-a hierarchy and two instances of *container* (maybe *container1* and *container2*) in a model fragment. The containers in the model fragment are instances of the generic concept *container* in the is-a hierarchy.

Attributes

Attributes are found in model fragments, scenarios and rules and represent structural relations between instances. The word *structural* must be read as the opposite of behavioral here; the structural description of a system is that part of the description that does not change in the course of a simulation. Attributes can be of topological nature, such as *the container contains liquid*, but may also be more abstract; such as *person A is a colleague of person B*.

Attributes can also describe a *property* of an instance. This type of attribute does not relate instances to one another, but relates an instance to a particular value out of a set of nominal values. *The container is open* is a typical example. There is some discussion about whether this type of

attribute can really be considered part of the structural description of a system, because property values do change in a simulation. This question is not relevant in the context of this document, however, and we will consider properties to be part of the structural description of a system.

Quantities and quantity spaces, values and derivatives

Quantities are variables that potentially change over time. In model fragments and scenarios, quantities are associated with instances of entities. In rules, however, they may exist on their own. A quantity has a set of possible *values* and a set of possible *derivatives*; both represented by a *quantity space*. In a simulation, quantities represent the dynamic behavior of a system.

The term *quantity* is used to refer to both the generic concept and an actual instance. In a model fragment or scenario, a quantity is declared with both a generic name (such as *height*) and an instance name (such as *height1* or *height of water column*). However, there is no place in a GARP model where all generic quantities are declared. In other words, there is no equivalent for quantities to the is-a hierarchy for entities. There is no need to formally list all generic quantities because the GARP language is declarative: it will recognize the generic quantities as unique concepts. It must be noted that in rules, the generic name of a quantity may be omitted to represent the fact that the rule will hold for any type of quantity.

A quantity space contains a set of possible qualitative values of a quantity on an ordinal scale. The values in a quantity space must alternate between *points* and *intervals*. An example of a quantity space is the set of qualitative states for the temperature of a substance. That quantity space could contain the interval *below melt point*, the point *melt point*, the interval *between melt point and boil point*, the point *boil point* and the interval *above boil point*.

Quantity spaces are defined separate from quantities; two quantities may use the same quantity space. This does not mean that the values of the quantities are equal; it means that both quantities use the same kind of value-set. Even if two quantities use the same quantity space and have the same qualitative value, the *quantitative* value of the quantities may not be the same. For example, if two boilers contain water and alcohol, which both have a quantity temperature that is at boil point, the qualitative values are the same, but their quantitative values are still different.

The possible *derivatives* of a quantity are also captured in a quantity space. The quantity space for derivatives contains the values *min*, *zero* and *plus*. In theory, derivatives can be represented by a different quantity space, but in practice the min-zero-plus range is always used.

Dependencies

Dependencies are relations between quantities, values or derivatives. They specify the constraints that hold between quantities, and are used in model fragments, scenarios and rules. There are four types of dependencies: inequalities, correspondences, proportionalities and influences¹.

Inequalities are statements of inequality or equality between quantities or between a quantity and one of its values or derivatives. They allow statements like *the value of quantity q1 is greater than the value of quantity q2* or *the value of quantity q1 is smaller than zero*. Inequalities also make qualitative calculus possible through the subtypes *plus* and *min*. This allows a modeler to specify that, for example, *the flow rate of water through a pipe is equal to the water pressure at one end of the pipe minus the water pressure at the other side of the pipe*.

Inequalities may refer to point values, but not to interval values. This is because qualitative values refer to the underlying quantitative value of a quantity, and interval values do not have a specified quantitative value, but rather represents a set of values. To clarify this with an example: using an inequality statement it is not possible to declare that the temperature of a substance is equal to the interval between its melt point and its boil point.

¹ Bredeeweg (1992) specified a fifth type of dependency: the implication. Implications relate two dependencies in a conditional statement: if the conditional dependency holds, then the implied dependency will also hold. In practice, implications are not used and we will not describe them here.

Correspondences state that particular values of two different quantities always occur together. There are two types of correspondences: value correspondences and quantity space correspondences. The latter can be seen as a set of value correspondences for each value in the quantity spaces of two quantities. It follows that a quantity space correspondence can only exist between two quantities with equal quantity spaces.

Correspondences can be directed or undirected. In a directed correspondence, one value can be derived from the other, but not the other way around. In an undirected correspondence, both values can be derived from the other.

Proportionalities are relations between the derivatives of two quantities. Proportionalities are always directed: if the influencing quantity increases, the influenced quantity also increases, or decreases. There are two types: positive and negative. A positive proportionality states that if the influencing quantity increases (or decreases), the influenced quantity also increases (or decreases). The negative proportionality states the reverse: if the influencing quantity increases (or decreases), the influenced quantity decreases (or increases).

Influences capture situation in which the value of one quantity influences the derivative of another quantity. For example: if the *flow rate of water* coming out of a *tap* is positive, the *level* of water in the *container* beneath it will start to increase (its derivative becomes *plus*). Two types of influences exist: positive and negative. In a positive influence, a positive value of the influencing quantity causes a positive derivative of the influenced quantity, and a negative value causes a negative derivative. In a negative influence, a positive value of the influencing quantity causes a negative derivative of the influenced quantity, and a negative value causes a positive derivative.

Model fragments

Model fragments allow a modeler to create reusable descriptions of subsystems of the real-world system described in a model. Model fragments are used to capture the notion of behavioral aspects that emerge from structure. For example, a model fragment could state that whenever an entity *liquid* exists, that liquid will have a quantity *amount*, the value of which is greater than *zero*. Another model fragment could state that whenever a *liquid* is contained in a *container* (i.e. the container has a relation *contains* with the liquid), then a quantity *level* exists, which is also greater than *zero*. These examples show both the reusability of model fragments (the first model fragment could be reused in the second) and the ability of model fragments to describe behavior emerging from structure (amount emerges in the first model fragment as a result of an entity liquid being present, and level emerges in the second as a result of the liquid being contained by the container).

A model fragment consists of a set of conditional elements (referred to as the *conditions*) and a set of consequential elements (referred to as the *consequences* or *givens*). The conditions are used to describe under which circumstances the model fragment applies, and the givens describe what can be assumed true if the conditions are met. The step ‘find applicable model fragments’ in Figure 3-1 should now be clear: the simulator looks for model fragments whose conditions match the current system description, and when it finds one it adds the elements in the givens to the system description. The elements in the conditions can be instances of entities, attributes, quantities, values and derivatives, inequalities and other model fragments. The givens can additionally contain correspondences and causal dependencies but cannot contain model fragments.

The declaration of a model fragment in GARP may also include the specification of a model fragment type or one or more parent model fragments. There are five model fragment types:

- **Single description model fragments** describe the static properties of a single element. The model fragment that states that when a *liquid* exists, that liquid has a quantity *amount*, given as example above, is a single description model fragment.
- **Composition models fragments** describe the static properties of a more complex system, a system consisting of several elements in a structure. An example of a composition model is a model fragment describing that if a *liquid* is contained in a *container*, that liquid has a quantity *level*.

- **Decomposition model fragments** describe the structure and workings of a complex system in terms of *part of* relations. In practice, this type of model fragment is rarely used because it is not often needed in a model and because decomposition can also be represented in a composition model fragment.
- **Process model fragments** describe processes, such as the flow of matter or energy as a result of an inequality. For example, heat will flow from a heat source to a boiler until the temperature of the heat source and the boiler are equal. A model fragment capturing this is a process model fragment.
- **Agent model fragment** describe the effect of influences from outside of the system. The *level of liquid* in a *container*, for example, can start to increase because a *tap* above the container was opened. In a model fragment describing this situation, the tap plays the role of agent: it initiates a process that does not result from an inequality.

We can think of these five types as the base of a hierarchy of model fragments. A model fragment may either descend directly from a model fragment type or it may descend from a parent model fragment. And because a model fragment can have more than one parent, multiple inheritance is possible.

There is a subtle but important difference between parent model fragments and nested model fragments. A nested model fragment, one that is found in the conditions block of a model fragment, is a model fragment that must be active in order for the model fragment in which it is contained to be applicable. Parent model fragments, on the other hand, place a model fragment in a hierarchy. This allows a modeler to create more specific versions of existing model fragments. As an example of a parent-child relation between two model fragments, a model could contain a model fragment *substance flow* that describes the flow of a substance in general terms, and a model fragment *gas flow* that describes properties specific to a flow of gas. In this case, the *gas flow* would be a child of the *substance flow*. In its simplest form, the *gas flow* model fragment would state that the entity *substance* in the *substance flow* must be a *gas* in order for the *gas flow* model fragment to be applicable².

An example of model fragment nesting is a *contained liquid* model fragment that has a nested *open container* model fragment. In this case, the nested model fragment states that the container in *contained liquid* must be an *open container*.

Scenarios

Scenarios describe an initial state in a simulation. Originally called input systems, scenarios describe the input to the simulator, based on which the simulator will search for applicable model fragments and rules. We can think of the scenario as ‘state zero’.

Scenarios closely resemble the conditions block of a model fragment, the exception being that a scenario does not contain nested model fragments. Scenarios can thus contain instances of entities, attributes, quantities, values and derivatives and inequalities.

Transition rules

Transition rules determine when and how a system moves from one state to a next state. They are applied in the second step of a simulation process: after the simulator has found and applied model fragments, it searches for applicable transition rules. The output of this step is a set of possible states. See Figure 3-1 for a reminder.

In many simulators, transition rules are considered to be part of the simulator itself. GARP takes a different approach here and explicitly includes the transition rules in the model. This gives the modeler the flexibility of creating domain-specific rules. In practice, however, this is not often done. Most transition rules can be described in domain-independent terms and GARP users usually copy the rules from an existing model when creating a new one.

There are three types of transition rules: termination rules, precedence rules and continuity rules. They are described below.

² This assumes that *gas* is a child of *substance* in the entity hierarchy.

Termination rules describe how a change to a quantity's value or derivative causes a state to terminate, and what will change in the next state. An example of a termination rule is: *if a quantity is increasing it will reach it a higher value in the next state*. Termination rules capture the notion of changes to a system over time.

A termination rule has a set of conditions, describing when the rule applies, and a set of givens, describing what the consequences are if the rule applies. The conditions may contain instances of entities, attributes, quantities, values and derivatives, inequalities and correspondences. In practice, however, a termination rule rarely contains entities or attributes. The givens may contain all the same elements the conditions does, and may additionally contain influences and proportionalities.

Precedence rules contain information about the ordering of termination rules. In situations in which multiple termination rules apply to a system description, precedence rules are used to determine which terminations have precedence over others and which may be merged into one termination. Two members of a precedence rule are the two termination rules that it applies to. A precedence rule may additionally contain instances of entities, attributes, quantities, values and derivatives, inequalities and correspondences. These elements and the two terminations describe the conditions under which the rule applies. Furthermore, a precedence rule has an *action*, which can be either *remove* or *merge*. The action describes the consequence of the rule: what happens to the two terminations that the rule applies to. If one of the terminations should have precedence over the other, the action is *remove* and the rule states which of the two terminations should be removed in favor of the other. If the two terminations are to be merged, the action is *merge*.

An example of a remove rule is one that states that if one termination exists in which a quantity is moving from a point value to an interval and another termination exists in which a quantity moves from an interval to a point, then the former will occur first. This is because an increase or decrease from a point value leads to a new value immediately, while an increase or decrease in an interval may take time.

Continuity rules describe how behavior in one state continues in the next. They apply to all elements in a system description that are not affected by a termination rule. An example continuity rule is that *if a quantity is increasing in one state, it will still be increasing in the next state or will have become stable*. A continuity rule has a set of conditions and a set of givens. The conditions may only contain values and derivatives of quantities. The givens may additionally contain inequalities.

States and state transitions

As explained in section 3.1, the output of the simulator consists of a set of states. States describe a possible state of behavior of the system; one that the system may reach at some point in time. State transitions describe transitions from one state to another. They capture why and how a state-change occurs.

In the first step of the simulation process, the simulator matches the scenario against the library of model fragments and adds the consequences of the matching fragments to the state description. In the second step it applies transition rules, which may also result in adding elements to the state. The result is a full state description containing all the entities, attributes, quantities, values and derivatives, dependencies, and model fragments that are active in the state.

State transitions represent the result of the second step in the simulation process: applying transition rules. They give a full description of which rules were applied, what caused them to be applied and what the results of applying the rules were. Transitions have a single from-state attribute and a set of to-state blocks. A to-state block contains a cause block (containing the rules that were applied), a conditions block (containing the model ingredients based on which the rules were applied) and a results block (containing the results of applying the rules).

Looking at the states and state transitions together, the simulator output can be viewed as a *state graph*. This graph emerges from the states themselves and the *from-state* and *to-states* attributes of the transitions. States are referenced by their *state number*. Figure 3-2 shows an example of a state graph. The initial state in a state graph is always the scenario on which the simulation was based.

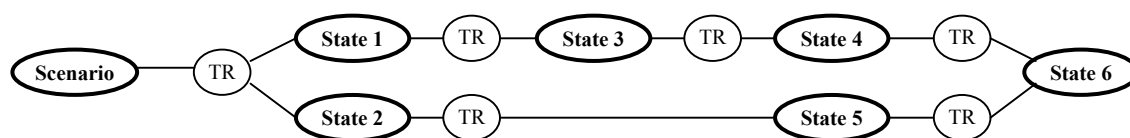


Figure 3-2: A simple state graph.

3.3 An example model: the u-tube

Throughout this chapter, most of the examples given refer to a model that has been used as a prototypical qualitative model for years. To clarify some of the concepts used in this chapter, and because the u-tube example is often used in this document, this section describes the u-tube model in greater detail.

Figure 3-3 depicts a u-tube as envisioned in the u-tube model. It consists of two containers filled with liquid and a path connecting the two containers through which liquid may flow. The containers are open, which means that the containers will overflow rather than explode when the level of liquid in them reaches the height of the container. The liquids have the quantities amount, level, and pressure. In Figure 3-3, the u-tube is in a state in which the level of liquid 1 is higher than the level of liquid 2. We can expect liquid to start flowing from left to right in the next state.

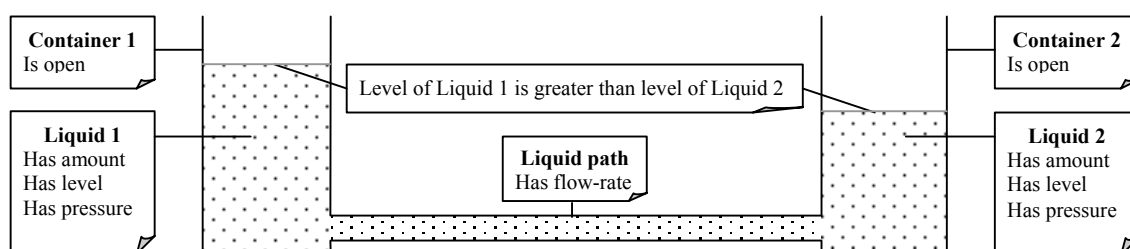


Figure 3-3: The u-tube.

Analysis of the u-tube domain

There are three distinct types of entities in this example: *container*, *liquid* and *liquid path*. A modeler creating a model of this u-tube could put only these three entities in the entity hierarchy, but the model becomes more articulate by including a distinction between *objects* and *substances* in the hierarchy. Figure 3-4 shows the entity hierarchy of the u-tube model. Substance is shown to have three children: solid, liquid, and gas. Only liquid is relevant in the example at hand, the other two are used in other examples in this document.

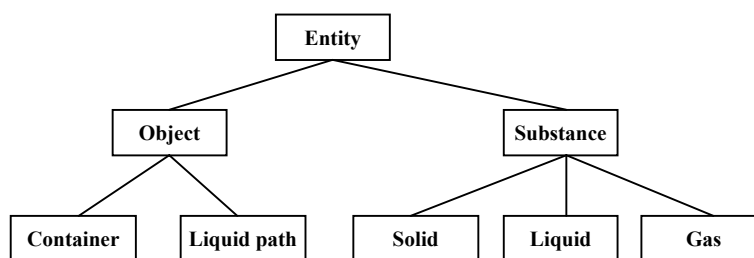


Figure 3-4: The entity hierarchy of the u-tube model.

In the u-tube domain, containers can be open or closed and they can contain liquid. Note that these are general statements about containers, not statements about one specific container: containers as a class of things can have the attributes *openness* (open or closed) and *contains* (an entity of type liquid).

In a u-tube, the liquid in a container has three relevant quantities: its *amount*, its *level* in the container and its *pressure* at the bottom of the container, at the level of the liquid path. Each of these quantities can be in three qualitative states: *zero* (when no liquid is present in the container), *plus* (when some liquid is in the container) and *max* (when the container is full). The modeler thus has to define a *zero-plus-max* quantity space.

A liquid path has a quantity *flow-rate*. For simplicity, the modeler assumes that a positive flow-rate is a flow of liquid in one direction and a negative flow-rate is a flow in the opposite direction. A flow-rate of *zero* means no flow at all. The quantity space *min-zero-plus* captures this. This is incidentally also the quantity space used for derivatives of quantities.

Having defined the entities and quantity spaces present in the domain, the modeler can now start creating a number of model fragments. In the process of analyzing the domain, the modeler has also discovered the attributes and quantities that are to be used in the model. These will be used in the model fragments.

The 'liquid view' model fragment

In analyzing the domain, the modeler took note of the fact that liquids have a quantity amount. This is the sort of information that is captured in single description model fragments, which describe the properties of a single entity. The liquid view model fragment is a single description model fragment and states the following:

If an entity of type liquid exists, that entity will have a quantity amount and the value of that amount will be greater than zero.

Translating this to a model fragment: *liquid view* has an entity of type *liquid* in the conditions block. In the givens block it has a quantity *amount*, which belongs to the liquid and has a quantity space of *zero-plus-max*. The givens block also contains a dependency *greater*, relating *amount* to the value *zero* in its quantity space.

The 'open container view' model fragment

Open container view is another single description model fragment. It contains only two model ingredients: an entity of type *container* and an attribute *openness* with value *open*. Both are part of the conditions block; the givens block is empty. This model fragment is used to state:

An open container is a container that has an attribute openness with value open.

Note that *open container view* is used for a different purpose than *liquid view*. *Liquid view* is used to derive new information (a quantity amount) from existing information (a liquid being present). *Open container view* is only used to create a structural description.

The 'contained liquid' model fragment

With a view on liquids and containers defined, the modeler can now talk about how liquid behaves inside a container. Liquid in a container has three quantities: *amount*, *level* and *pressure*. These quantities are related to each other: the amount determines the level and the level in turn determines the pressure of the liquid at the bottom. This is captured in the *open contained liquid* model fragment. *Open contained liquid* is a composition view model fragment: it contains a number of entities in a structure and describes the behavior of that structure.

At this point the modeler can reuse the model fragments built so far by including them in the conditions block of the new model fragment. Instead of creating a new open container in this model fragment (by adding a *container* and an attribute *openness* to the conditions block), the modeler adds a container and states that it be like the one described in the *open container* model fragment. Similarly, the modeler adds a liquid to the conditions block and states that it be like the one described in the *liquid view* model fragment. Finally, a *contains* relation is added between container and liquid. The conditions block of the *open contained liquid* model fragment now states:

If a container exists that adheres to the conditions put forth in the open container model fragment, and a liquid exists that adheres to the conditions put forth in the liquid view model fragment, and the container contains the liquid, then apply the conditions of this model fragment.

We can thus think of nested model fragments as a set of extra conditions for the model fragment they are embedded in. For the simulator, a model fragment with nested model fragments applies only if the nested model fragments also apply. Using nested model fragments also means that the modeler does not have to redefine the model ingredients in the givens blocks of the nested model fragments. This is because they are added to the system description when the simulator determines that the nested model fragments apply. In the *open contained liquid* model, this means that the modeler does not have to include the quantity *amount* and its dependency *greater* in the givens block, because these are

already defined in the *liquid view* model fragment. The modeler can concentrate on the behavior that follows from the structure of the *open contained liquid* model fragment. Which is that:

- Two new quantities of the liquid emerge: *level* and *pressure*. Both use quantity space *zero-plus-max*, as does quantity *Amount* (which was introduced in the liquid view model fragment).
- There is a directed *quantity space correspondence* and a *positive proportionality* between *amount* and *level*. The quantity space correspondence states that if amount is at value v , then level will be at that same value in its quantity space. The proportionality states the same thing for the derivatives of amount and level: when amount has derivative d , so does level.
- There is a directed quantity space correspondence and a positive proportionality between *level* and *pressure*.

The modeler now has all the building blocks needed to construct the u-tube as depicted in Figure 3-3.

The liquid flow model fragment

In the *liquid flow* model fragment the modeler reuses the *open contained liquid* model fragment twice and connects the two containers with a liquid path. Reusing the open contained liquid model fragment two times yields two containers: *container 1* and *container 2*, both filled with liquid: *liquid 1* and *liquid 2*. To this, the modeler adds a *liquid path* and two *connected* relations that state that the path is connected to both containers. The result of this structure is described in the conditions by four statements:

- *Liquid path* has a quantity *liquid-flow* with quantity space *min-zero-plus*.
- The value of *liquid-flow* is equal to the *pressure* of *liquid 1* minus the *pressure* of *liquid 2*.
- *Liquid-flow* has a *negative influence* on the *amount* of *liquid 1*.
- *Liquid-flow* has a *positive influence* in the *amount* of *liquid 2*.

In other words: if there is a pressure difference between liquid 1 and liquid 2, then a liquid-flow will occur which is either positive (from container 1 to container 2) or negative (from container 2 to container 1) and which changes the amounts of the liquids.

Simulating the u-tube

The model fragments described above describe what conclusions the simulator can draw from certain structures. Model fragments are abstract descriptions of domain knowledge. To instantiate them, the simulator needs a scenario.

A scenario in the u-tube domain could state that:

- Two containers exist: *container A* and *container B*. Both are *open* and both contain liquid: *liquid A* and *liquid B*.
- A *liquid path* exists. It is *connected* to *container A* and to *container B*.
- The *level* of *liquid A* is *greater* than the *level* of *liquid B*.

Based on this scenario, the simulator will quickly derive that:

- Because *liquid A* and *liquid B* exist, there must be a quantity *amount* for both, and the values of these amounts are greater than *zero*. This follows from applying the *liquid view* model fragment.
- Because *liquid A* and *liquid B* are contained in an open container, the liquids must have the quantities *level* and *pressure*. This follows from applying the *open contained liquid* model fragment.
- Because the level of each liquid has a directed quantity correspondence with amount (according to the *open contained liquid* model fragment) and because both amounts are greater than zero, both levels must also be greater than zero.
- Because the pressure of each liquid has a directed quantity correspondence with level (again according to the *open contained liquid* model fragment) and because both levels are greater than zero, both pressures must also be greater than zero.

- Because two open containers filled with liquid exist that are both connected to a liquid path, a quantity liquid-flow exists. The value of liquid flow is equal to the pressure difference between liquid A and liquid B. This follows from applying the *liquid flow* model fragment.

Note that the simulator does not know the actual values for the levels of liquid A and liquid B. It does know that both are greater than zero, cannot be greater than max (the highest value in their quantity spaces) and that the level of liquid A is greater than the level of liquid B. From this the simulator concludes that either both levels have value *plus* or level A has value *max* and level B has value *plus*³. Either way, the simulator will eventually find that the pressure of liquid 1 must be higher than the pressure of liquid 2, and that this situation will trigger a liquid flow from container A to container B. This flow will cause the amount of liquid A to decrease and the amount of liquid B to increase, which will eventually cause the levels of the two liquids to become equal. At that time, the pressure difference between the two liquids becomes zero and the flow stops.

This example illustrates the strength of qualitative reasoning. The u-tube model describes what happens in a u-tube in all-qualitative terms; no quantitative values for the quantities are needed at any time. Not only can the simulator reason with the qualitative description of the domain, it can also serve as a tool for providing causal accounts of what is happening.

3.4 Conclusion

The previous chapter, chapter 2, focused on the use of knowledge articulation tools as educational instruments and explored the possibilities to facilitate the development of these tools by creating a module that offers knowledge-modeling functionality. In that chapter, qualitative reasoning was chosen as an appropriate language for knowledge articulation in the GKOM module. This chapter has introduced GARP as the language that GKOM will be compatible with.

Offering a knowledge articulation language is only one of the features identified in chapter 2. The next chapter returns to these features and outlines the design of the GKOM module.

³ They cannot both have value *max*, because max is a point value. Two quantities cannot both be at a point and be unequal at the same time.

4 GKOM design

This chapter describes the overall design of the GKOM module. The first section explains GKOM's place in the modeling process. Apart from giving an architectural overview, this section shows that the functionality that GKOM offers is both functionally and architecturally divided over a number of modules. In section 4.2, these modules are briefly introduced. Section 4.3 describes the knowledge-capturing module in greater detail.

4.1 GKOM and the modeling environment

Figure 4-1 shows a high level overview of the modeling environment in which GKOM operates. The modeling environment for which GKOM is designed is one in which the GKOM module and the user interface module together form a stand-alone application (the modeling application), which may or may not be connected to remote services and other modeling applications. In the figure, the modeling application is connected only to a simulation service.

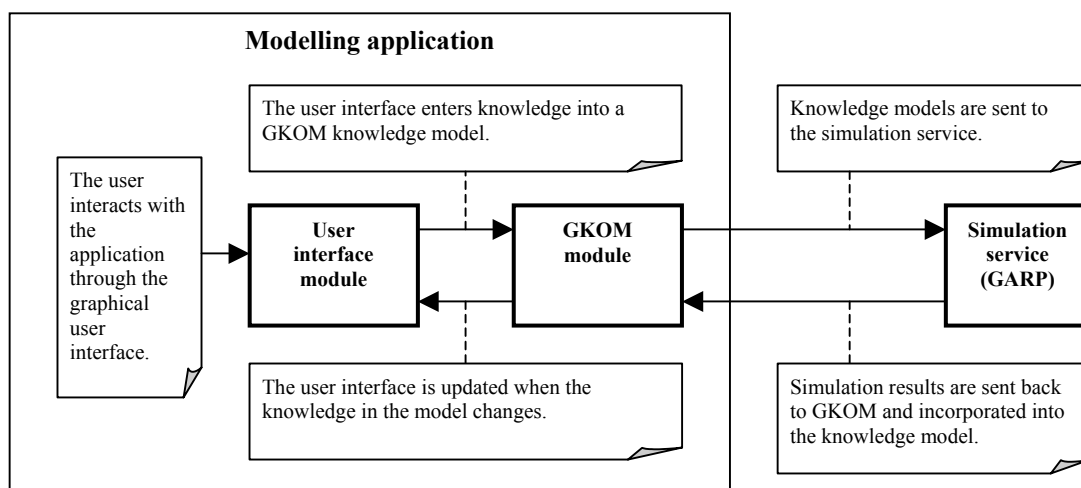


Figure 4-1: Architecture of a modeling environment.

Before looking at the GKOM module at a greater level of detail, it is important to realize that the GKOM module must be independent of both the user interface module and any remote services. Regarding the user interface module, the idea is to create a module that handles all of the modeling functionality and leaves only the representational issues to the application builder. This entails that no assumptions may be made about how the knowledge is presented to the modeler. An application builder may choose to only use a subset of the functionality offered. Regarding the remote services, the GKOM module is designed not to depend on remote services for its operation. Also, GKOM is only partially dependent on how the communication between the GKOM module and remote services is implemented and on the knowledge format used by the simulator. This is achieved by creating sub-modules inside the GKOM module that handle the communication and the import and export of GKOM models, as the next section explains.

4.2 Sub-modules of the GKOM module

Figure 4-2 shows a high-level overview of the inner workings of the GKOM module. The tasks that are to be performed by GKOM are carried out by a set of sub-modules.

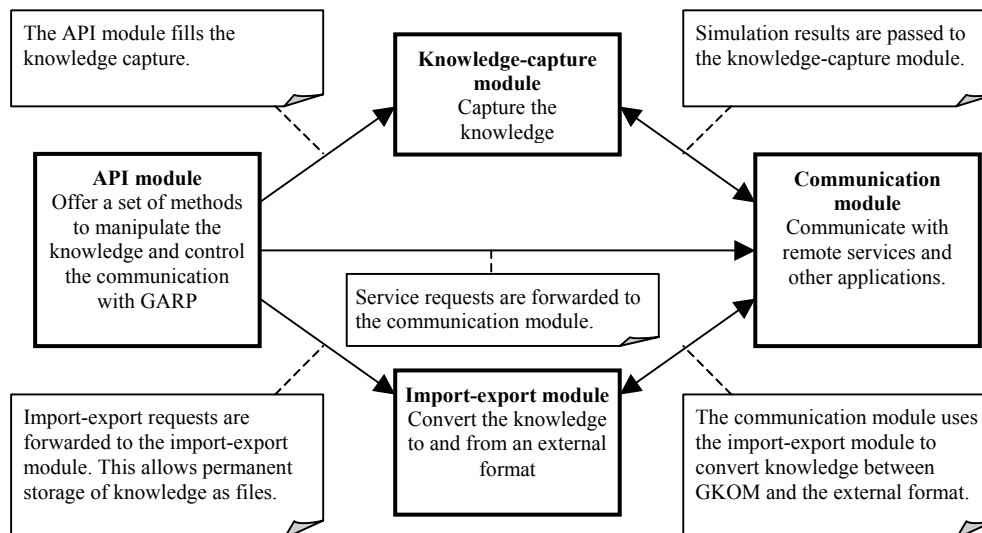


Figure 4-2: Sub-modules of the GKOM module and the communication between them.

These sub-modules are:

- The Application Programming Interface (API) Module. The task of this module is to offer the user interface module, depicted in Figure 4-1, a set of handles by which to control the knowledge captured and control the simulation process carried out by the GARP server. Because the GKOM module will be incorporated into a modeling application, these ‘handles’ are in the form of method calls on GKOM objects. The full set of these methods is referred to as the Application Programming Interface.
- The knowledge-capturing module. It is the task of this module to capture the actual knowledge that a modeler is working on.
- The import-export module. This module converts the knowledge representation used by the GKOM module to a textual representation that can be sent over a network connection or written to a file.
- The communication module. The current task of this module is to act as a client to the simulation service. This includes sending and receiving knowledge models and instructing the simulation service to run simulations on the knowledge.

This separation into modules is a conceptual one. In particular, the API module and the knowledge-capturing module are closely intertwined in GKOM. The other two modules are modules in a literal sense: they are strictly separate from the GKOM module as a whole and can easily be replaced. The import-export module can be replaced when the knowledge format required by external services changes. The communication module can be replaced when the communication protocol to control the simulation service changes.

The knowledge-capturing module

The knowledge-capturing module has two tasks: it captures knowledge in the form of a model, and it offers modeling support. The knowledge representation itself uses the vocabulary discussed in chapter 3, with some extensions discussed in section 4.3. The issue of modeling support is discussed in chapter 5.

The API module

An API is a module’s interface with the outside world. The API defines the functionality of a module, while hiding the complexity of the actual implementation of that functionality. Broadly, there are two categories of functionality that GKOM offers: functionality to create and manipulate a model and functionality to communicate with external sources. One such source is a simulation service; simulation is not done by GKOM itself, but by the GARP server, to which GKOM serves as a client. It forwards all simulation requests to the server and incorporates all the simulation results directly into the knowledge model.

The import-export module

The GKOM knowledge representation is an object-oriented one, consisting of objects with properties and methods. These objects exist in the computer's memory. To communicate a model with external services (or even to store a model as a file on a disk) a representation is needed that can be sent over a network connection or written to a file. The task of the import-export module is that of converting from the 'native' GKOM representation to an external one, and from the external representation back to the native representation.

An important requirement for this module is that it can be easily extended – or replaced entirely – when the external format changes. In fact, it is not inconceivable that multiple external representations will one day exist concurrently. The GKOM module must thus be independent of which import-export module is used, what the external knowledge representation looks like and how that external representation is created. But GKOM will include a 'default' import-export module, which translates to and from the external representation used by the simulation service. Currently, this is a representation in the eXtensible Markup Language (XML, e.g. Harold & Scott Means, 2002).

The communication module

The task of the communication module is to handle all communication between the GKOM module and external services or other modeling applications. It is controlled by the API module and uses the import-export module to convert GKOM models to external representations and external representations to GKOM models. Like the import-export module, the main requirement for this module is that it should be independent of the other modules; the GKOM module as a whole must be able to operate without it or with a different communication module. It must be possible to alter the communication module when changes are made to the protocol used to communicate with remote sources, and these changes should not affect the operation of the GKOM module.

Presently, only one remote source exists: a simulation service. It is implemented as a GARP application embedded in a HyperText Transfer Protocol (HTTP) server¹. Traditionally, the clients of HTTP servers have been browsers, requesting HTML pages on the server. In the case of the GARP server the communication module in GKOM is the client and the 'pages' it requests are virtual, representing instructions (such as 'load model', 'run full simulation' etc) to the simulator. The server responds with the result of the instruction, which could either be a report ('model loaded', for example) or the output of a simulation process.

4.3 Design of the knowledge representation

This section outlines the design of the knowledge-capturing module. It describes the primitives of the GKOM knowledge representation and the design decisions made in the development of GKOM. Because the GKOM module is written in Java, this section begins with a very short introduction to object-oriented programming to familiarize the reader with the object-oriented terminology.

4.3.1 A very short introduction to Object Oriented Programming

Object Oriented Programming (OOP, e.g. Satzinger & Ørvik, 1996) has originally been developed at the Xerox Palo Alto Research Center in the 1970'ties. The procedural languages of the time basically consisted of long sequences of instructions to the processor, operating on a set of data shared by the entire program. The Xerox researchers felt that it would be more natural to have a language that would consist of objects: small pieces of data, accessible only by sending a message to the object that requests the data from it. The objects would be representations of real-world things (a person, for example) and the messages would return properties of these things (like a person's date of birth).

A programmer writing in an object-oriented language creates *classes* that consist of *members* (also referred to as *properties*) and *operations* (also called *methods*). Staying with the example of the representation of a person, a very simple class Person would consist of a property 'name' and a property 'date of birth'. It would have operations that return these properties ('getName' and

¹ HTTP version 1.1 is described in RFC 2616, which can be found at <http://www.ietf.org/rfc/rfc2616.txt>.

‘getDateOfBirth’, for example) and operations that perform some sort of logic on these properties (like a method ‘getAge’ that uses the current date and the date of birth property to determine how many years have passed in between).

Classes are the blueprints for *objects*; an object is an *instance* of a class, created in a running program. We could have one instance of Person with name ‘Mary’ and date of birth ‘May 15, 1974’ and one with name ‘John’ with date of birth ‘January 6, 1962’. This distinction between classes and objects is a crucial one, and one that is often hard to grasp for novice object-oriented programmers. Classes describe the *types* of objects that can exist in a program and are created by the programmer, while objects come into existence at runtime and carry the actual data associated with a single instance of the class. Two things add to the confusion about classes and objects. Foremost, the term object is often also used to refer to the class. Second, the code that the programmer writes inside a class always operates on instances of classes. For example, a piece of code could iterate over a set of Person-objects to find all Persons above the age of 65.

We include this swift introduction to OOP to clarify that the distinction between the API module and the knowledge-capture module made above is a purely conceptual one. Knowledge is captured in GKOM as a set of instances of classes. Example classes include Entity, Quantity, and ModelFragment. The API available to the application builder is made up of the operations defined for these classes. Example operations are Entity.getName, Quantity.getValue and ModelFragment.getConditions.

4.3.2 Three knowledge levels, ten modeling primitives

The GKOM knowledge representation offers ten modeling primitives: entities, properties, relations, quantities, quantity spaces, values, derivatives, dependencies, model fragments and scenarios. Additionally, GKOM discriminates between three types, or levels, of knowledge: type-level, occurrence-level, and instance-level knowledge. The knowledge in a model is separated in three types of knowledge to underscore the fact that different parts of a model play different roles. The first role is that of type or class. Modeling primitives that play this role generically define the existence of certain concepts. An entity at the type-level, for example, represents a class of entities. The second role is that of occurrence. Occurrences are members of the classes defined at the type-level and modelers use them to construct model fragments and scenarios. The third role is that of instance, or simulation result. The subtle but important difference between occurrences and instances is that occurrences are created and controlled by modelers and used to build a model, whereas instances are created as the result of a simulation process.

The knowledge levels and knowledge primitives are the two dimensions along which the knowledge representation is constructed: with a few exceptions, each of the knowledge levels contains all of the knowledge primitives. In other words: there are separate classes for all primitives at each of the levels, yielding 30 unique classes. A separate class represents the model itself. Class Model serves as a container for primitives at each of the knowledge levels. In a model, primitives at the type-level represent the ontology of the domain that the model captures. They prescribe how primitives at the occurrence-level, used to create model fragments and scenarios, can be used. Primitives at the instance-level represent simulator output.

The type-level

Primitives at the type-level represent ontology-like knowledge. At this level a modeler specifies what types of things exist in the domain and how these things are related. The distinction between types and occurrences (the second level) is like the distinction between classes and instances in the object-oriented paradigm: types are the blueprints for occurrences. Using the u-tube model outlined in section 3.3 as an example, the type-level of that model will contain, among other things, the entities shown in Figure 3-4: ‘entity’, ‘object’, ‘container’, ‘fluid path’, ‘substance’, and ‘liquid’. It will contain the relations ‘contains’, between container and liquid, and ‘connected’, between container and fluid path. And it will contain the quantities ‘level’, ‘pressure’, and ‘amount’, all related to liquid, and the quantity ‘liquid flow’, related to the liquid path.

It is important to realize that these type-level primitives are used to describe an ontology of the domain, not an actual construct like a model fragment or scenario. In fact, the type-level objects in a

model define what occurrence-level objects are available for building model fragments and scenarios, and how these can be used. This is described in detail in the next section.

There are a number of reasons to include type-level knowledge in the knowledge representation:

- The type-level objects can be used to create a toolkit-like functionality in a modeling application. In an educational setting, a teacher may predefine the type-level primitives, and the task for the students may be to use them as building blocks to create model fragments and scenarios.
- The information recorded at the type-level can be used for offering support. The support system can use that information to determine whether modelers make proper use of the building blocks offered to them, and correct them when they do not. In this sense, creating the type-level of a model can be seen as ‘teaching’ the support system about the domain. Offering support is the subject of chapter 5.
- The type-level primitives capture the assumptions that a modeler makes about a domain. Forcing the modeler to express these promotes reflection. So does confronting the modeler with the consequences of these assumptions when at some later time the modeler tries to use an occurrence in a way that is not allowed by the ontology, or, in other words, by the modeler’s own assumptions.

At the same time, however, the use of a type-level adds an extra layer of complexity to the knowledge representation. It adds semantical richness, but does so at the expense of what we could call freedom of expression. A modeler can no longer simply introduce an entity while building a model fragment, for example. Instead, the modeler must first create the entity type and place it in the entity hierarchy, to subsequently use an occurrence of the newly created entity type in the model fragment. And this is true for all modeling primitives, up to a new value for a quantity.

To answer the question of whether the benefits of using type-level knowledge justify the loss of freedom of expression, consider two possible applications that may use the GKOM module. One is an application in which students express their knowledge of a domain. In this kind of application, all the benefits mentioned above are clear and the type-level knowledge would be used to make the application easier to use, to offer good learner support and to provoke contemplation on the part of the learner. Another possible use of the GKOM module, albeit not one that is given much attention in this document, is in a modeling environment for experts. Experts will expect to be treated as such and may indeed consider the type-level as an extra layer of complexity that does not add to the expressiveness of the knowledge representation. Even though the third of the three benefits mentioned may still apply to them. But for such an application, the application builder can, at least in part, automate the process of creating the type-level. For example, when the expert introduces a new entity in a model fragment, the application could simply create a new entity type and a new occurrence of that entity type at the same time². This would eliminate the objection, and the benefits still hold.

The occurrence-level

Objects at the occurrence-level are the building blocks with which model fragment and scenarios are composed. They represent how the prototypical elements at the type-level are used to create new, aggregate types. The occurrence-level is also subject to the bulk of the support built into GKOM.

A type-level object describes a class of things and the properties that members of that class may have. An occurrence describes an individual member of the class and the actual properties it possesses, along with their values. Taking the model fragment ‘liquid view’ in section 3.3 as an example, the entity ‘liquid’ in that model fragment is an occurrence of the entity type ‘liquid’. The entity type describes liquids conceptually: a liquid is a particular type of substance and it may have the quantities ‘amount’, ‘pressure’, and ‘level’. The entity ‘liquid’ in the model fragment ‘liquid view’ specifies that this particular liquid has a quantity ‘amount’.

For entities, this difference between conceptual descriptions and individual descriptions is familiar to most knowledge engineers. GKOM extends this notion to all modeling primitives: all primitives are represented at both the type-level and the occurrence-level. Returning to the example

² This raises the issue of how much can be automated. Where in the entity hierarchy is the new entity type to be placed? What if an entity by the given name is already present there? This issue will be discussed in chapter 9.

model, the quantity ‘amount’ in the ‘liquid view’ model fragment is an occurrence of a quantity type ‘amount’ in the ontology, as the relation ‘contains’ between the entities ‘container’ and ‘liquid’ in the ‘contained liquid’ model fragment is an occurrence of a relation type ‘contains’ in the ontology. An occurrence cannot exist without a type and cannot have more than one type. A type can have an unrestricted number of occurrences (including none) in one or several model fragments or scenarios.

An occurrence must be used in accordance with what its type specifies about possible use. An entity at the occurrence-level, for example, may have no other quantities than those possessed by its type. And a relation at the occurrence-level may have no other left and right hand side than the ones its type has³. This is why we refer to the type-level as a model’s ontology: it describes what elements exist in a domain and how these elements may be used to create aggregates. This does not mean that an entity in a model fragment or scenario *must have* all quantities or properties that its type has, it means that the entity *may not have* any other ones. The type-level does not mandate a particular kind of use; it prohibits invalid use.

The support system actively monitors the construction of model fragments and scenarios and intervenes when the modeler uses an occurrence in an invalid way. For this task it uses the information at the type-level. An intervention by the support system forces the modeler to either build the model fragment or scenario differently, or update the type-level information. Either way, the result is that the type-level accurately describes the domain ontology at any time.

All modeling primitives are represented at the occurrence-level except one: the scenario. This stems from the fact that occurrence-level objects are used to create model fragments and scenarios, and thus include only the primitives that can be used in these aggregates. That does not include scenarios. It does include model fragments, because model fragments can be used inside other model fragments, as the example model in section 3.3 demonstrates.

The instance-level

The objects at the instance-level represent the result of a simulation process. Instances are similar to occurrences in that they represent individual members of a class rather than concepts, and in that they have a reference to a type. The difference between them is largely a conceptual one: it underscores the fact that occurrences are created by the modeler, and instances by the simulator. But there are also two technical differences. The first is that the validity of instances with respect to their types is not actively enforced by the support system. This is because instances are not created by human agents but by the simulator⁴, and the simulator can be expected to produce valid output. The second difference is that instances are state-aware. Every instance knows in which states in the simulation it exists, and some instances, like quantities and properties, have values that may change from state to state.

The modeling primitives

The ten modeling primitives in GKOM are derived from a subset of the GARP terms introduced in section 3.2. They are introduced briefly here, and are discussed at length in chapters 7 and 8.

The modeling primitives are:

- **Entity** A thing or concept. At the type-level, an entity describes a class of things, at the occurrence and instance-levels an individual thing. Entities can have properties, relations, and quantities, and at the type-level an entity can have a parent (a more general entity) and children (a set of more specific entities).
- **Property** An attribute of an entity. In GARP this primitive is called attribute, which also includes relations. In GKOM these are two separate primitives. At the type-level, a property represents a possible attribute of an entity and has a set of possible values. At the occurrence-level a property has a single, actual value and at the instance-level it potentially has a different value in each state. A property is related to a single entity.
- **Relation** A relation between two entities. In GARP this primitive is called attribute, which also includes properties. A relation has a left hand side entity and a right hand side entity and exists at all knowledge levels.

³ The inheritance of properties, relations and quantities provides some flexibility in this respect, as explained in section 4.3.3.

⁴ More precisely, instances are created by the input-output module translating the output of the simulator to GKOM.

- **Quantity** A variable associated with an entity. A quantity has a set of possible values, contained in a quantity space, and a set of possible derivatives. At the occurrence-level, a quantity may additionally have an actual value and actual derivative, which must be one out of the set of possible values and derivatives. At the instance-level a quantity potentially has a different value and derivative in each state. A quantity may be associated with a number of entities at the type-level, and with a single entity at the occurrence and instance-levels.
- **Quantity space** A set of alternating point and interval values, representing the possible values of a quantity. At the type-level there is a many-to-many relation between quantities and quantity spaces: a set of quantities may use the same quantity space and a single quantity may use several quantity spaces. At the occurrence and type-levels, a quantity space is always associated with a single quantity.
- **Value** The value of a quantity. A value is part of a quantity space and represents either a point or an interval. Every value is associated with one quantity. Values exist at all three knowledge levels.
- **Derivative** The direction in which a quantity changes. Technically, derivatives are exactly the same as values in GKOM; the difference between the two is entirely conceptual.
- **Dependency** A binary relation between quantities, values, or derivatives. Dependencies are further divided into inequalities, correspondences and causal dependencies, the latter including proportionalities and influences, which were introduced separately in section 3.2. Dependencies only exist at the occurrence and instance-levels, because the full set of dependency types is fixed, while including a type-level representation would suggest that the modeler could create custom dependency types.
- **Model fragment** A construct describing the emerging properties of an aggregate of other primitives. Model fragments exist at all three knowledge levels.
- **Scenario** A description of an initial state of a system, serving as input to the simulator. At the type-level a scenario represents the input to the simulator, at the instance-level it describes the state of the input system in a particular state of the simulation. Scenarios do not exist at the occurrence-level because they cannot be used inside a model fragment or another scenario.

The statement at the beginning of this section that all modeling primitives exist at all three levels can now be clarified. The type-level consists of the classes `EntityType`, `PropertyType`, `RelationType`, `QuantityType`, `QuantitySpaceType`, `ValueType`, `DependencyType`, `ModelFragmentType`, and `ScenarioType`. Types are related to other types, so `EntityTypes` have `PropertyTypes`, `RelationTypes` and `QuantityTypes`, `QuantitySpaceTypes` consist of `ValueTypes` and are associated with a number of `QuantityTypes`, and `RelationTypes` relate two `EntityTypes`, to give a few examples.

The occurrence-level contains the classes `Entity`, `Property`, `Relation`, `QuantitySpace`, `Value`, `Derivative`, `Dependency`, and `ModelFragment`. For brevity, these names are not appended with ‘Occurrence’. Every object of a class at the occurrence-level is related to (is an occurrence of) an object of the class at the type-level that represents the same modeling primitive. So every `Entity` is related to an `EntityType`, every `Property` to a `PropertyType`, etc. Besides that occurrences refer only to other occurrences, so a `Property` belongs to an `Entity`, a `Relation` has a left and right hand side `Entity` and a `Value` is part of a `QuantitySpace` and belongs to a `Quantity`.

The instance-level contains the classes `EntityInstance`, `PropertyInstance`, `RelationInstance`, `QuantityInstance`, `QuantitySpaceInstance`, `ValueInstance`, `DerivativeInstance`, `DependencyInstance`, `ScenarioInstance`, and `ModelFragmentInstance`. Any object of one of these classes has a reference to an object of one of the classes at the type-level, so every `QuantityInstance` is related to (is an instance of) a `QuantityType`, every `ValueInstance` to a `ValueType`, etc. Other than that, instances refer only to other instances.

Another statement made above, that type-level objects prescribe how the occurrence-level objects can be used, can also be clarified now. It means that an occurrence `O1` can be related to an occurrence `O2` if the type of occurrence `O1` is related to the type of occurrence `O2` in the same way. So for example, a modeler can only create a property `P` for entity `E` if `E` is an occurrence of `EntityType` `ET` and `P` is an

occurrence of PropertyType PT and PT belongs to ET. Whether PT belongs to ET is defined in the *entity hierarchy*, discussed in the next section.

4.3.3 The role of the entity hierarchy

At the type-level, entities are organized in a hierarchy. Every entity has a single parent and an unrestricted number of children. Because only entity *types* are organized this way, the entity hierarchy is more formally also referred to as the *entity type hierarchy*.

GARP also has the notion of an entity hierarchy, an example of which was given in Figure 3-4. But the GKOM entity hierarchy has an additional purpose: it enables inheritance of properties, relations and quantities. Any child entity type CET of parent entity type ET inherits all property types, relation types, and quantity types from ET and can additionally be given a set of property types, relation types, and quantity types of its own. In the u-tube example the entity type ‘substance’ would be given the quantity type ‘amount’, so that all children of substance (‘solid’, ‘liquid’ and ‘gas’) inherit that quantity. Quantities that apply only to one of the children are assigned to that particular child, so ‘level’ would be assigned to ‘liquid’, not to ‘solid’ or ‘gas’.

This principle of inheritance influences what is considered *valid use* of objects at the occurrence-level. The question at the end of the previous section, when does a property type PT belong to an entity type ET, can now be answered: PT belongs to ET if PT was assigned to ET directly or if ET inherits PT from its parent. This is also true for relation types and quantity types. In the U-tube example this entails that occurrences of entity types ‘solid’, ‘liquid’ and ‘gas’ can be given occurrences of quantity type ‘amount’, while only occurrences of ‘liquid’ can be given an occurrence of ‘level’.

GKOM uses this extended notion of the entity hierarchy for two reasons. First, giving property, relation and quantity types a place in the entity hierarchy transforms the type-level from a collection of unrelated concepts into an ontology of the domain. In other words: the type-level does not only enumerate all possible entities, properties, relations and quantities in the domain, it also contains information about how all of these elements are related. Incorporating ontology-like information into a model is desirable in itself, for the reasons given in section 4.3.2. The second reason is related to the fact that GKOM will not allow the modeler to create duplicate property, relation or quantity types and at the same time will not allow the modeler to assign a property type to more than one entity type or a relation type to more than two (left and right hand side) entity types. This forces the modeler to consider carefully where to place properties and relations in the entity hierarchy, since their placement will determine which occurrences of entity types may use them. We expect this to lead to greater conceptual clarity of the model.

4.3.4 Model, simulation, state, and transition

There are four classes in the GKOM knowledge representation that cannot be considered to belong to the type, occurrence or instance-levels. These classes represent the model itself, the states and transitions resulting from simulations based on a model and the concept of a simulation itself. All of these classes can be considered containers for other objects.

- **Model** All model ingredients described in this chapter are in some way part of a model. Class Model captures this fact. A model consists of a set of entity, property, relation, quantity, quantity space, model fragment and scenario types. These types are *direct members* of a model. Objects of other primitive types are *indirect members* of a model: occurrences through the model fragment and scenario types of a model, and instances through the simulations based on a model.
- **Simulation** A simulation is a collection of states and state transitions based on a particular scenario. Simulations are stored along with the model for future reference. An object of the Simulation class holds a reference to the ScenarioType that it is based on and a set of State and Transition objects.
- **State** A state holds all the instance-level objects that are active in a particular state of the simulation. Objects of the State class are members of a Simulation.

- **Transition** A transition captures how one state changes into the next state. Objects of the state class are members of a Simulation and contain all instances involved in a state-change.

4.4 Conclusion

This chapter has given a high-level overview of the design of the GKOM module. The GKOM module consists of three sub-modules: the knowledge-capture module, the import-export module and the communication module. In the remainder of this document, the primary focus will lie on the knowledge-capturing module. It has been introduced extensively in this chapter. The next chapter focuses entirely on one of the most important features of that module: offering modeling support.

5 Offering support

This chapter describes the modeling support built into the GKOM module. This first section briefly introduces why support is necessary, what sort of operations require support, what sort of mistakes a modeler might make and how the GKOM module will prevent these mistakes. After this introduction, a detailed description of the operations that require support is given in sections 5.2 through 5.4. These sections handle the creation, modifications and removal of knowledge ingredients, respectively. Finally, section 5.5 handles the technical implementation of the support system.

5.1 Introduction.

Building a model is the process of articulating one's knowledge in a particular formalism. Users of a modeling environment can be expected to be familiar with the domain for which they are building a model, but not necessarily with the modeling formalism. Modeling support should thus focus on the correct use of the modeling language, rather than on the semantics of a model. The errors that a modeler can make in creating a GKOM model, fall into three categories:

- Errors related to the use of duplicate names.
- Errors related to conflicts between information at the type and occurrence-levels.
- Errors related to unforeseen side effects of actions.

The actions of a modeler also fall into three categories:

- Actions that result in the creation of new model ingredients (discussed in section 5.2).
- Actions that modify existing model ingredients (section 5.3).
- Actions that remove model ingredients from a model (section 5.4).

Duplicate name errors occur when a modeler uses the same name for different model ingredients in the same context. They can happen while creating new model ingredients or while modifying the names of existing ones. Conflicts between information at the type and occurrence-levels are conflicts between a model's ontology and the contents of its scenarios and model fragments. The ingredients of scenarios and model fragments are occurrences of the types defined in the ontology, and they can only be used in ways defined in the ontology. Errors of this kind occur when creating knowledge and modifying knowledge. Unforeseen side effects of actions can occur when a user is modifying or removing model ingredients. Side effects are conflicts between model ingredients at the type and occurrence-levels that result from operations upon different model ingredients.

Preventing these errors is the first goal of the GKOM support system, and it does so by not allowing them to happen. The second goal is to explain the error to the modeler. The modeler needs to become familiar with the modeling formalism, and errors are a good opportunity for GKOM to explain the formalism to the user. GKOM must explain why the suggested action is not allowed and what the effects would be if the action would be carried out. The following three sections cover creating knowledge, modifying knowledge and removing knowledge, and show how these types of actions can lead to the three types of errors described above.

5.2 Creating knowledge

Guiding the creation of knowledge in a model is not only a matter of checking conditions that must apply. It is primarily a matter of not offering operations that would create knowledge in improper ways. A scenario, for example, cannot contain a model fragment, so the GKOM API does not define a method to add a model fragment to a scenario. Only knowledge creation operations that create valid model ingredients are defined in the GKOM API. This way, the API implicitly supports the modeler in not creating a knowledge model that is structurally incorrect. The GKOM support system is thus concerned with operations that are correct in certain circumstances, and incorrect in others.

A modeler can create model ingredients at two levels in GKOM: the type and occurrence-levels. At both levels, knowledge creation can result in duplicate names. Additionally, at the occurrence-level errors related to conflicts between the type and occurrence-levels can occur.

5.2.1 Creating elements at the type-level

One operation that can have a valid effect in some situations and an invalid one in another is the operation that gives a model ingredient a name. The result of that operation will only be valid if the name given is unique. Users can distinguish between the different model ingredients in a model based on two characteristics of a model ingredient:

- The type of model ingredient. A user knows the difference between an entity and a relation, for example.
- The model ingredient's name. The name identifies the model ingredients among the other elements of the same type.

The combination of these two characteristics should be unique for every model ingredient in a model. Not enforcing this restriction would lead to an ambiguous model, both from the perspective of the user (who would not be able to distinguish between the two elements) and from the perspective of the simulator (who would simply consider them the same).

However, the above does not imply that no two elements of the same type can exist in the same model. This is because all elements in a model exist in a particular context. Entities in the entity hierarchy exist in the context of the model that the entity hierarchy is part of. This entails that no two entities may exist in the entity hierarchy by the same name, as explained above. But entities in a model fragment exist in the context of that model fragment. It is not an error to have two entities with the same name in two different model fragments, nor is it an error to have an entity in a model fragment that has the same name as an entity in the entity type hierarchy. We can think of the context that a model fragment provides as a 'name space' for its members.

In short, no two model ingredients of the same type with the same name may exist in the same contexts. The following contexts are found in a model:

- The model itself. The model is the context for the entities, properties, relations and quantities at the type-level, the quantity spaces, the scenarios and the model fragments. It follows that no two entities, properties, relations, quantities, quantity spaces, model fragments or scenarios may exist in a model by the same name, but it is possible that two members of different types have the same name in a model. For example: it is possible that an entity and a model fragment have the same name.
- Each of the model fragments and scenarios. A model fragment or scenario enforces the same restrictions upon its members as a model does: all members of the same type must have a different name, but it is all right for two members of different types to have the same name. But each of the model fragments and scenarios enforce these restrictions individually. Because of this, it is possible that two members of the same type in two different model fragments or scenarios have the same name. Also, the context provided by a model fragment or scenario is not related to the context provided by the model. Members of a model fragment or scenario may have the same name as members of the model.
- Each of the quantity spaces. Values exist in the context of a quantity space. This means that no two values by the same name may exist in a single quantity space, but two different quantity spaces may both contain a value v .

To prevent the modeler from using duplicate names, GKOM performs a check every time a user creates a new model ingredient and refuses to create the element if an element of that type by that name already exists in the context that the modeler is in. This achieves the first goal of offering support: preventing mistakes. To achieve the second goal – explaining the user about the formalism – the error text presented to the user should make clear that it is the context that prevents the user from creating the new element. As an example, consider a modeler creating a model fragment *liquid flow*, in which an entity *container* exists and to which the modeler is adding another entity *container*. This is an error: within one context (the *liquid flow* model fragment) the user is creating two elements of the same type (both entities) with the same name. To resolve the error, the user could change the context,

change the type of element being created or change the name of the element. The user interface module could choose to offer all three resolutions to the modeler. But because it is unlikely that the modeler would like to change the context or the type of element being created, the user interface could simply present the modeler with an error text that could read:

An entity called 'container' already exists in model fragment 'liquid flow'.

This makes it clear that it is the name 'container' that causes the problem and that it is the model fragment 'liquid flow' that vetoes the creation of the new element.

When creating model ingredients at the type-level, using duplicate names is the only type of error that GKOM needs to check for.

5.2.2 Creating elements at the occurrence-level

At the occurrence-level, the GKOM support system checks for both duplicate name errors and errors related to conflicts between information at the type-level and information at the occurrence-level. Duplicate name errors are described in the previous section and are no different at the occurrence-level: they occur when the modeler creates two elements of the same type by the same name in the same context. At the occurrence-level, the context is a scenario or model fragment. This section focuses on inconsistencies between the information at the type-level and the information at the occurrence-level.

Recall from the previous chapter that members of model fragments and scenarios are of a different type than members of a model. The members of a model form an ontology of the model's domain: they define the types of entities that exist in the domain, the kind of properties and quantities that these entities can have and the relations they can be part of, and the types of quantity spaces that the quantities may use. In a sense, they provide the building blocks for creating model fragments and scenarios.

Members of model fragments and scenarios are occurrences of the types defined in the model. The entities *container 1* and *container 2* in model fragment *liquid flow*, for example, are occurrences of the generic entity *container* in the model. This generic entity is a template for the containers in the model fragment: it defines how *container 1* and *container 2* can be used in the model fragment (they can contain a liquid, for example), but not how they will be used. How they will be used is up to the modeler creating the *liquid flow* model fragment.

It is GKOM's task to ensure that the model ingredients at the occurrence-level adhere to the ontology defined by the model ingredients at the type-level; GKOM will not allow any action that causes inconsistencies between the information at these two levels. To implement this functionality, GKOM requires the modeler to provide a type for every model ingredient created at the occurrence-level. Whenever the modeler tries to do something with the newly created element that is not explicitly allowed by its type, GKOM explains to the modeler that he is creating an invalid occurrence and refuses to perform the action.

An example of this behavior is when a modeler is creating a model fragment that contains a container – an occurrence of the entity type *container* in the entity hierarchy. At the type-level, the modeler defined that a container can have a property *openness* and a relation *contains* with entity *liquid*. Suppose that an unrelated property at the type-level is *status*, which is used for pipes and can have the values *aligned* and *unaligned*. Then, the modeler tries to assign the property *status* to the container in the model fragment. To do this, the modeler must supply the property type *status* (the property as defined at the type-level) and the occurrence of the container used in the model fragment. The modeler's intention is to add a property *status* to the entity of type *container*. But this action is invalid, because containers do not have a property *status*, according to the ontology. GKOM would thus refuse to fulfill the modeler's request and respond by saying:

Entities of type 'container' cannot have a property 'status'.

With this behavior, GKOM achieves both goals of offering support. By refusing to perform the request, GKOM ensures that the model does not become invalid. With the explanation text, it explains the modeler about the formalism: no inconsistencies may exist between the information at the type-level and the information at the occurrence-level.

Relations and quantities at the occurrence-level are created in a comparable manner. To create a relation as part of a model fragment or scenario, a modeler must supply a relation type (a member of the model) and a left hand side and right hand side entity from the model fragment or scenario. GKOM checks whether the entity types of the entities in the model fragment are the same as those placed at the left and right hand side of the relation's type. If this is not the case, creation of the relation is aborted and with an appropriate explanation. To create a quantity as part of a model fragment or scenario a modeler must supply a quantity type and an entity from the model fragment or scenario that the quantity will belong to. Before actually creating the quantity, GKOM checks whether the type of the entity supplied can have quantities of the specified quantity type. If this check fails, the quantity will not be created and a text is generated explaining why the action would create an invalid quantity.

Before a value can be assigned to a quantity in a model fragment or scenario, the quantity must be given a quantity space. The modeler defines the types of quantity spaces as members of the model and assigns these quantity space types to quantity types as possible quantity spaces of the quantity. When a modeler subsequently assigns a quantity space to a quantity in a model fragment or scenario, the quantity space given must be one of those defined as the possible quantity spaces for the quantity. If this is not the case, the action is aborted and the user will be told something like:

'min-zero-plus' is not a valid quantity space for quantities of type 'level'.

Once a quantity in a model fragment or scenario has been assigned a quantity space, the quantity has a set of possible values. These can then be used as the actual values of the quantity (the value of quantity q is plus) or in dependencies (the value of q is greater then zero). But the modeler may not use values that are not part of the quantity space of a quantity. If a user would try to assign the value 'min' to a quantity 'level' that has a quantity space with the values 'zero', 'plus' and 'max plus', GKOM refuses to assign the value and tells the modeler:

'min' is not part of the quantity space of 'level'.

To summarize, creating new model ingredients can yield two types of errors: those related to naming and those related to conflicts between types and occurrences. The next section, which covers modifying model ingredients, introduces the third type of error: model ingredients becoming invalid as a side effect of operations on other model ingredients.

5.3 Modifying knowledge

Modeling is a process of refinement; early versions of a model rarely capture the domain correctly. Refining is an important part of the modeling process and GKOM must support modifying models properly.

Modifying model ingredients can lead to all three types of errors described in the introduction. Renaming an element is no different then giving it its initial name: it can result in a duplicate name error. Moving a property, relation or quantity at the occurrence-level from one entity to another, can result in a conflict between the type and occurrence-levels in the exact same way that assigning a newly created property, relation or quantity to an entity can. These two types of errors have been discussed in sections 5.2.1 and 5.2.2.

Errors specific to modifying knowledge are those that cause inconsistencies in a model. For example, if the modeler changes the quantity space of a quantity in a model fragment or scenario, any values of that quantity that are part of the model fragment or scenario could become invalid as a side effect. Consider a quantity in a model fragment or scenario that has been assigned the value *min* and is subsequently given a quantity space that does not contain the value *min*. In that case, changing the quantity space would make values in the model fragment or scenario invalid. The invalid values are a side effect of an operation that does not concern the values directly; the operation is concerned with quantity and its quantity space, not with the values in the model fragment.

Another example is when a modeler deletes model ingredients at the type-level that have occurrences at the occurrence-level. When a modeler deletes an entity type from the entity hierarchy,

but entities in model fragments or scenarios reference that entity as their entity type, the entities in the model fragments or scenarios become invalid as a side effect of the operation.

GKOM does not allow any action that makes a model invalid. This entails that none of the individual model ingredients in a model may become invalid. This supports the modeler in the sense that the model cannot ‘break’. But simply telling the modeler that the suggested action is not allowed is not sufficient. What is of interest to the modeler is what the consequences would be of the suggested action: what model ingredients would become invalid as a side effect if the action would be carried out. This allows a modeler to inspect these model ingredients and fix the potential problem (potential because the operation is not actually carried out) and then try again.

The next sections describe the type of modifications that can be made on model ingredients and the inconsistencies that they can result in.

5.3.1 Changing the structure of the entity hierarchy

At the type-level, entities, properties, relations and quantities are used to define an ontology of the domain. It is this ontology that specifies how members of model fragments and scenarios – occurrences of the types defined in the ontology – can be used, and GKOM uses the ontology to support the modeler in creating members of model fragments and scenarios, as described above. Modifying the elements that make up the ontology thus means changing the definition of what is valid use of model ingredients at the occurrence-level. When these changes are made *after* creating occurrence-level elements, these occurrence-level elements could become invalid; it could lead to configurations that GKOM would not have allowed at creation time.

Making the modeler aware of such inconsistencies will help the modeler to develop a more profound understanding of the domain. To achieve this, support for modifying knowledge must not be confined to simply not allowing the modeler to do things that make the model invalid. More importantly, the modeler must be given insight into *what* the effects of modifications would be and *why* these modifications would lead to an invalid model. The following four sections show how this is achieved.

At the type-level, all entities have a *parent entity* and a set of *child entities*. Through these attributes the entities define an entity hierarchy. Properties, relations and quantities at the type-level all refer to entities in some way: a property has a single entity to which it belongs, a relation has entities as left and right hand side and a quantity has a set of entities to which it belongs. Because of this, properties, relations and quantities can be viewed as members of the entity hierarchy: they are placed alongside the entities they belong to. From this point of view – which is illustrated in Figure 5-1 – changes to entities, properties, relations and quantities are changes to the structure of the entity hierarchy.

5.3.1.1 Moving entities around in the entity hierarchy

When an entity at the type-level is given a new parent, that entity moves from one place in the entity hierarchy to another. The result of this action is that the entity will stop inheriting the properties, relations and quantities from its old parent and start inheriting the set of properties, relations and quantities from its new parent. Figure 5-1 illustrates this process. The figure shows an entity type hierarchy and the properties, relations and quantities that each entity has. The modeler is about to move entity E4 from the right branch of the hierarchy to the left branch of the hierarchy.

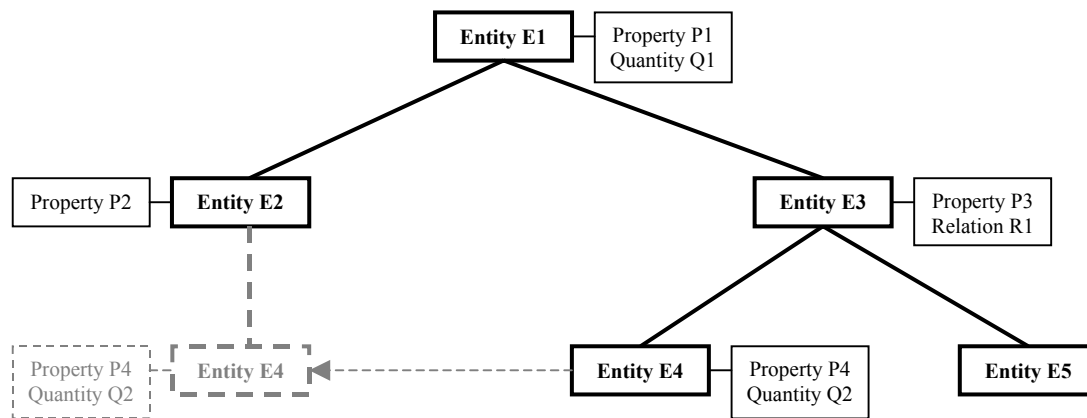


Figure 5-1: Moving an entity from one place in the hierarchy to another. Entities are highlighted.

As a child of E3, E4 inherits property P3 and relation R1 from entity E3, and property P1 and quantity Q1 from entity E1. Additionally, E4 itself has property P4 and quantity Q2. When E4 is moved, it will take P4 and Q2 with it, but it will no longer inherit property P3 and relation R1. The status of property P1 and quantity Q1 remains unchanged from the point of view of entity E4, because it will inherit these through its new parent E2. It will also begin inheriting property P2.

If entity E4 has any occurrences in the model (that is: if any entities are used in model fragments or scenarios that are of type E4) and any of these occurrences have a property of type P3 or are part of a relation of type R1, then moving the entity in the hierarchy will lead to an inconsistent model. GKOM will not allow this. When a modeler requests an entity in the hierarchy to be moved, GKOM will do the following:

1. Determine which properties, relations and quantities are no longer valid in the new situation.
2. Iterate over the occurrences of the entity and select all those that use any of the properties, relations and quantities selected in step 1.
3. If any occurrences were selected in step 2, the move operation is cancelled and the user is told that the operation would lead to an inconsistent model. The modeler is given the set of occurrences selected in step 2 for inspection.

5.3.1.2 Moving a property type from one entity type to another

Properties at the type-level have an entity-attribute: the entity they belong to. It is this attribute that places the property in the entity hierarchy, and changing it results in moving the property to another place in the entity hierarchy. Figure 5-2 illustrates this process.

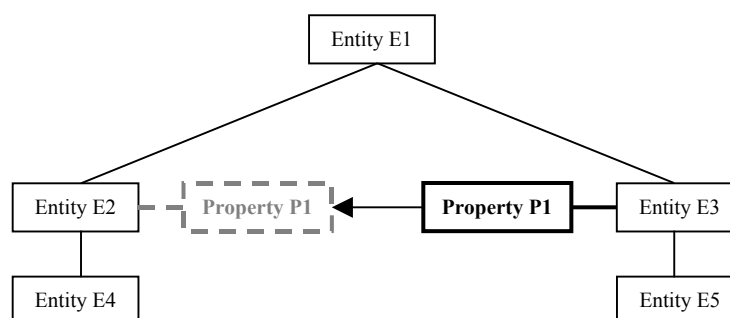


Figure 5-2: Moving a property from one entity to another. Properties are highlighted in this figure.

Before the move operation, property P1 belongs to Entity E3. This means that any occurrence of E3 may have a property of type P1. Entity E5 inherits P1 from E3, so any occurrences of E5 may also use the property. Moving property P1 to entity E2 leads to a situation in which occurrences of P1 can no longer belong to entities of type E3 and E5. In this example, this means that if P1 has any occurrences at all, these occurrences (properties in model fragments or scenarios) will become invalid¹.

¹ The fact that all occurrences of P1 become invalid is a result of the configuration used in this example. If property P1 would be moved to E5 instead of E2, only a subset of the occurrences of P1 would be affected.

GKOM will not allow the occurrences of P1 to become invalid. Before the actual move operation is carried out, GKOM iterates over the set of occurrences of P1 to see whether any of these belong to an entity of type E3 or any of its children. If so, GKOM cancels the operation and tells the modeler that the operation would lead to an inconsistent model. References to all occurrences of P1 that would become invalid are given to the modeler for inspection.

5.3.1.3 Changing the left and right hand side of a relation

Inheritance of relations down the entity hierarchy works in the same way as inheritance of properties does, but it is not the relation itself that is inherited, but one of the relation's sides. In the example depicted in Figure 5-3, entities E4, E5 and E7 inherit the left hand side of relation R1 from entity E2. Entity E6 inherits the right hand side of R1 from E3. Note that E7 is the right hand side of relation R2, and at the same time inherits the relation's left hand side from E5.

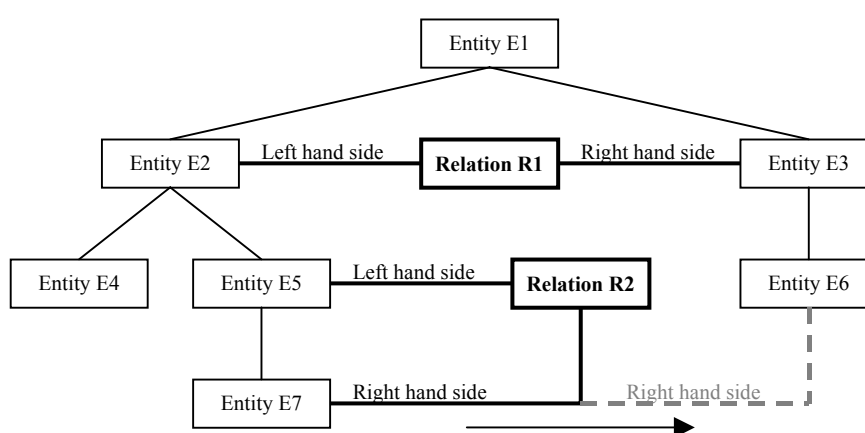


Figure 5-3: Changing the right hand side of a relation. Relations are highlighted in this figure.

The effect of changing the left or right hand side of a relation type is that occurrences of that relation type (relations in model fragments or scenarios) may become invalid. In model fragments or scenarios, the left and right hand side entities of relations are occurrences of the entities in the entity type hierarchy. Following the example in Figure 5-3, when the right hand side of relation R2 is changed from entity E7 to entity E6, all occurrences of R2 that have an occurrence of E7 as their right and side, will become invalid.

When a modeler changes the left or right hand side of a relation type, GKOM iterates over the occurrences of the relation to see whether they would become invalid. An occurrence would become invalid if its left or right hand side entity is an occurrence of the entity that will no longer be the left hand side or right hand side of the relation at the type-level. If any invalid occurrence is found, the operation is aborted. GKOM explains to the modeler that the operation would create an inconsistent model and passes the set of occurrences that would become invalid.

5.3.1.4 Changing the entities of a quantity

Quantities in the entity hierarchy exhibit the same behavior that properties and relations do, but have one unique feature: they can belong to several entities. Unlike properties (that belong to a single entity) or relations (that have a single left and right hand side), quantities can be used at multiple places in the entity hierarchy. This makes quantities easier to use: a modeler does not have to look for the proper place in the entity hierarchy to put a quantity (as would be necessary with a property), but can simply put the quantity wherever it is needed.

Figure 5-4 illustrates this unique feature of quantities. Quantity Q1 belongs to E3 and E4. Quantity Q1 belongs to a single entity: E1. Like properties and relations, quantities are inherited down the entity type hierarchy, so all entities in the figure 'have' quantity Q1. Quantity Q2 belongs to entities E3 and E5, but it is about to be removed from E5.

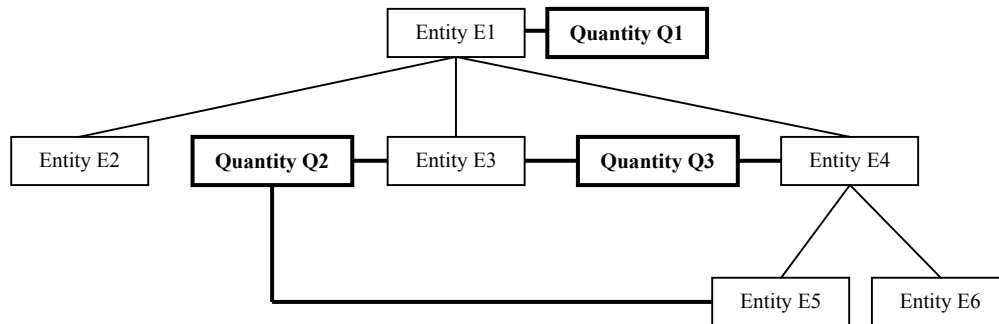


Figure 5-4: Quantities in the entity hierarchy and their multiple relations with entities.

Modifying the set of entities that a quantity belongs to can be done in two ways: by adding entities to the set and by removing entities from it. Adding an entity to the set cannot lead to any inconsistencies, because it can never make any of the occurrences of a quantity invalid. Removing an entity from the set can. Taking the situation in Figure 5-4 as an example, if entity E4 would be removed from the set of entities of quantity Q3, then all occurrences of Q3 that belong to occurrences of E4, E5 or E6 would become invalid.

GKOM will not allow this to happen, but it can tell the modeler exactly why the action cannot be performed. It will generate a list of all occurrences of quantity Q3 that belong to occurrences of entities E4, E5 and E6 and present that list to the modeler. The modeler can then choose to modify the occurrences or refrain from changing quantity Q3 in the first place.

5.3.2 Modifying quantity spaces at the type-level

In relation to quantity spaces at the type-level, two things must be monitored by the support system: changing the set of possible quantity spaces of a quantity type and modifying the set of values of an individual quantity space.

Figure 5-5 shows how quantity types and their occurrences are related to quantity space types and their occurrences. Every quantity is an occurrence of a quantity type. Quantity types can have a number of quantity space types. A quantity has a single quantity space, which is an occurrence of one of the quantity space types of its quantity type.

A quantity space type can belong to a number of quantity types. A quantity space at the occurrence-level, however, belongs to a single quantity. Every quantity has its own quantity space in GKOM, but it is possible (and in fact quite common) that several quantities use quantity spaces of the same type.

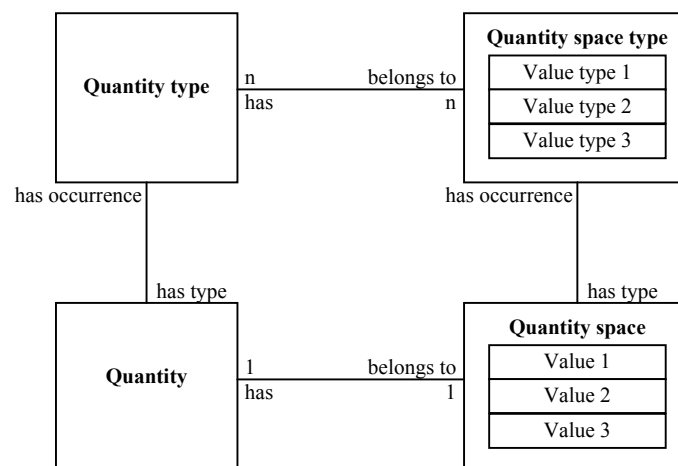


Figure 5-5: Dependencies between quantities, quantity spaces and their occurrences.

Changing the set of possible quantity spaces of a quantity is a process that resembles that of changing the possible entities of a quantity. Adding a quantity space to the set will never result in errors, but removing one might.

Trough the set of quantity spaces of a quantity, a modeler specifies what types of quantity spaces are acceptable for what types of quantities. This is part of the ontology of a model. When a modeler subsequently creates an occurrence of a quantity type, that occurrence can only be given one of the quantity spaces defined for the quantity type. When the set of quantity spaces for the quantity type is later changed, occurrences may exist that use the quantity space being removed.

GKOM will not allow this to happen. Instead, when a modeler removes a quantity space from a quantity at the type-level, it will generate the set of occurrences that use the quantity space and offer that set to the modeler.

Like most of the model ingredients in a GKOM model, quantity spaces exist at all three levels: the type, occurrence and instance-level. Quantity spaces at the type-level are part of a model's ontology and belong to quantities at the type-level, as described above. The occurrences of these quantity spaces are quantity spaces used by quantities at the occurrence-level (quantities in model fragments and scenarios) and the instances of these quantity spaces are those used by quantities at the instance-level (quantities in a simulation). Occurrences and instances of quantity spaces are created after their type: they contain the same set of values². These occurrences and instances must thus be kept in sync with their type.

Two kinds of changes can be made to a quantity space at the type-level: values can be added and removed. Adding a value will never result in any inconsistencies in the model, but removing one can. If the value that is being removed is in use, the action should not be allowed.

But the fact that occurrences of a quantity space exist does not imply that the values in that quantity space are actually in use. A value is in use when:

- A quantity exists in a model fragment or scenario that has been assigned that value.
- A dependency exists in a model fragment or scenario that has the value as its left or right hand side.

As an example of the second point: if a scenario exists with a dependency that states that the value of quantity q is greater than v , GKOM considers the value v of quantity q 'in use'. Removing v from the quantity space of q is then not allowed.

When a modeler tries to remove a value from a quantity space at the type-level, GKOM does the following:

1. It builds a set of quantities that use the quantity space by iterating over the occurrences of the quantity space asking them for the quantity they belong to.
2. It asks these quantities to what model fragment or scenario they belong.
3. It asks these model fragments and scenarios whether they contain the value that is being removed; either in their set of values (in which case the value is directly assigned to the quantity) or as left or right hand side of their set of dependencies.
4. It builds a list of all model fragments and scenarios that contain the value and passes this list to the modeler.

Normally when GKOM returns a set of model ingredients that would become invalid by a particular action, this set contains occurrences of the type that the modeler was modifying. In this case, that would be a set of quantity spaces at the occurrence-level. But that would be of little use to the modeler, who needs to know why the value cannot be removed, not in which occurrences of the quantity space it exists. Passing a list of quantities would be more appropriate, but that would lead the modeler to believe that it is the quantity vetoing the removal of the value, which is not the case. It is the model fragments and scenarios containing the value that would become invalid by removing it. Thus, returning a list of model fragments and scenarios is the best option.

² More precisely, an occurrence of a quantity space contains occurrences of the values in its type, and an instance of a quantity space contains instances of the values in its type.

5.3.3 Modifying scenarios and model fragments

This section deals with modifying model ingredients at the occurrence-level. Occurrence-level ingredients are members of scenarios and model fragments. As described in section 5.2, occurrences must be used in a way that adheres to the constraints defined by their types. This is true when creating members of scenarios and model fragments, and must remain true when these members are later modified. Sections 5.3.3.1 and 5.3.3.2 investigate the operations on occurrence-level elements that potentially make other elements invalid as a side effect. Two such operations exist: changing an entity's type (thus changing what properties, relations and quantities are valid for the entity) and changing a quantity's quantity space (thus changing what values are valid for the quantity).

5.3.3.1 Changing an entity's type

Entities in model fragments can be reused in other model fragments. A modeler can create a new model fragment as a child of another model fragment and reuse the entities from the parent. In such a case, the modeler can change the type of the reused entities to more specific ones, so that the child model fragment becomes more specific than the parent model fragment. This section describes what errors can result from changing an entity's type. Before doing so, it should be noted that the entity is the only occurrence-level element of which the type can be changed at all. None of the other occurrences allow their type to be changed, for two reasons. First, changing the type can affect the validity of the occurrence-level element. Changing the type of a relation, for example, can make the relation invalid in the context of the two entities that it relates. And changing the type of a quantity can make its quantity space invalid. The second reason is that changing an occurrence's type changes what it represents into something else. An example of changing the type of a property would be to go from a property 'aligned' to a property 'openness'. In the case of a relation, an example would be to go from a relation 'contains' to a relation 'connected'. Offering this sort of operation is not strictly necessary: a modeler who creates an occurrence of the wrong type can simply throw that occurrence away and create a new one. So while, from the modeler's perspective, the added benefit of being able to change the type of an occurrence is small, allowing it would require extensive support. Because of this, GKOM does not allow modelers to change the types of occurrences. But entities form an exception, because their types are embedded in a hierarchy. Suppose the entity hierarchy of a model contains an entity *substance* that has three children: entities *solid*, *liquid* and *gas*. A modeler may create a model fragment that describes some properties of *liquids*, and later realize that these properties in reality hold for all substances. In that case, the modeler will want to change the type of the liquid originally created to *substance*.

The problem with changing the type of an entity is that the validity of an entity's properties, relations and quantities depends on the entity's type. Figure 5-6 depicts a situation in which changing the type of an entity would lead to an invalid configuration. On the left is a fragment of the entity hierarchy. On the right is an entity *E1* in a model fragment or scenario that has a property *P1* and a quantity *Q2*. The entity is an occurrence of entity type *ET2* in the hierarchy, and *P1* and *Q2* are occurrences of property type *PT1* and quantity type *QT2*, respectively.

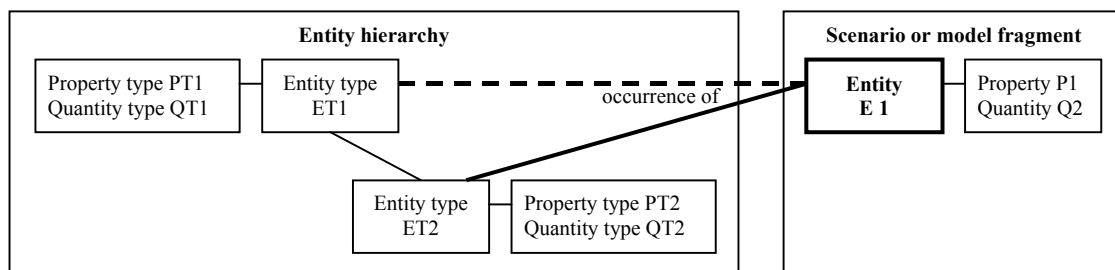


Figure 5-6: Changing the type of an entity.

Suppose the modeler tries to change the type of *E1* to entity type *ET1*. This is an invalid action, because *E1* has a quantity of type *QT2*, which belongs to *ET2*. *ET1* does not have this quantity type; so

changing the type of *E1* would make quantity *Q2* invalid. When a modeler changes the entity type of an entity, GKOM does the following:

1. It makes a list of all properties and quantities the entity has and all relations it is part of.
2. It removes from that list all properties, relations and quantities that will still be valid for the new type.
3. If any properties, relations or quantities remain in the list, it aborts the operation and passes the remaining elements to the modeler for inspection.

If the list is empty after step 2, the operation is carried out.

5.3.3.2 Changing the quantity space of a quantity

Two conditions must hold when changing the quantity space of a quantity in a scenario or model fragment: the quantity space must be a valid one for the quantity (it must be an occurrence of one of the quantity space types that belong to the quantity's type) and the new quantity space must at least contain those values from the old quantity space that are used by the quantity or by the scenario or model fragment that it is part of. The concept of 'usage' of values is explained in section 5.3.2.

The support for verifying the first of these conditions is described in section 5.2. Changing the quantity space of a quantity is no different from setting the initial one. But the second condition is different: once a quantity has been given a quantity space, values from that quantity space may have been assigned to the quantity and may have been used as left or right hand side of dependencies that are members of the scenario or model fragment that the quantity is part of. If this is the case, the old quantity space cannot be removed. Instead of changing the quantity space, GKOM will explain the problem to the modeler and pass the set of values that are in use for inspection.

5.4 Removing elements from a model

All model ingredients at the type-level may have occurrences in model fragments or scenario's. Once a type has an occurrence, removing that type would make the occurrence invalid. In order to be able to determine the consequences of removing a type from a model, all types know all their occurrences in GKOM. When a modeler tries to remove a type, GKOM will simply ask the type for its occurrences. If it has any, GKOM aborts the action and gives the modeler the list of occurrences that would become invalid. The modeler can inspect this list to determine what to do next.

For most elements at the type-level, the check for occurrences is sufficient to determine whether it can safely be removed because no other elements in the model depend on them. There is one exception: removing an entity type from the entity hierarchy. Entity types can have child entity types and properties, relations and quantities. Removing an entity impacts these elements. GKOM's normal behavior in this sort of situation is to not allow the removal and pass the modeler a list of elements affected. But for removing entity types from the entity hierarchy, GKOM offers an alternative way to handle inconsistencies: moving all dependent elements to the entity's parent. This process is as follows:

1. Take all the entity's properties and quantities and assign them to the entity's parent.
2. Take all relations that the entity is part of, find out whether the entity is the left or right hand side of the relation and assign that side to the entity's parent.
3. Take all of the entity's children and make them children of the entity's parent.
4. Take all of the entity's occurrences and set their type to the entity's parent.
5. Remove the entity.

There is one precondition to this process: the entity being removed must have a parent. This is true for all entity types except the top-level entity. When a modeler tries to remove that, the request will be denied.

At the occurrence-level, elements fall into three categories with respect to their possible removal: elements that can always be removed because no other elements can depend on them, elements that can only be removed after verifying that no other elements depend on them and elements that can simply not be removed. The latter is not really a category, but rather a single element: it is not possible to remove a value from a quantity space at the occurrence-level. Quantity spaces at the occurrence-

level contain the exact same values as their types (quantity spaces at the type-level) and these values change only when the type changes. Removing a value from a quantity space at the occurrence-level is an operation that is simply not supported.

The first category of elements, the ones that can always simply be removed, consist of properties, relations and dependencies. These would be removed from a model fragment or scenario. They can always be removed because no other elements can ever depend on them: properties and relations can be removed from the entities they belong to without affecting the entity and dependencies can be removed from a model fragment or scenario without affecting these constructs.

Entities and quantities make up the second category. Before removing them from the scenario or model fragment that they are part of, GKOM must assure that no other elements in the model depend on them. For example: removing an entity that has a property would leave that property without an entity it belongs to and make the property invalid. The entity may also have dependent relations and quantities, and in a model fragment the entity could be one that is reused in another model fragment (if the model fragment is a parent to, or appears as the nested model fragment of, another model fragment). Quantities in a scenario or model fragment may also have dependent elements: one of its values, assigned as the actual value of the quantity in that scenario or model fragment or dependencies of which the left or right hand side is either the quantity itself or a value of the quantity.

GKOM supports two ways of removing entities and quantities from a model fragment or scenario. One is the normal behavior: remove the entity or quantity only if it has no dependent elements, and if it does, abort the operation and return a list of all dependent elements. The alternative way is to simply remove all dependent elements along with the entity or quantity that is being removed. For quantities this means remove all values and dependencies that refer to the quantity. For entities it means remove all properties, relations and quantities that the entity refers to. Note that when removing the quantities that belong to an entity, the quantity's values and dependencies are also removed.

Removing an entity type from the entity hierarchy and removing an entity or quantity from a model fragment or scenario form an exception to the general rule that GKOM will not allow any operation that affects elements that are not themselves part of the operation. The methods for removing these three elements are unique in the sense that they offer a solution to the problem of dependent elements, whereas all other operations will simply report the problem and abort. There are two reasons for making this exception:

1. The operations are likely to be common ones that should be supported and not left to the modeler or user interface module.
2. The solutions to the problem of dependent elements are straightforward in these cases.

This is sufficient reason to make the exceptions. Also, although the rule of aborting when dependent elements exist is broken, the rule that no inconsistencies may be introduced in a model is not broken. Rather, possible inconsistencies are resolved.

5.5 Implementing support

GKOM is a passive module. It offers a set of functions that are called through an API and result in an internal representation of knowledge. But by design, it cannot actively call functions on the user interface module, because it explicitly does not make any assumptions about how the user interface is organized. This means that GKOM cannot take the initiative in making suggestions about the model. What GKOM can do is evaluate the effect of an action suggested by the modeler, and reject the action if it is invalid or would lead to an inconsistent model. It then has an opportunity to explain why the suggested action is invalid, thus teaching the user something about the formalism.

Java has a built in mechanism to abort the execution of a method, called the exception mechanism. In brief, it allows a method to declare that instead of running normally, it may abort execution by 'throwing an exception'. If a method is declared in this way, the caller of that method (which in the case of GKOM would be the user interface module) must declare how it will handle the exception if it occurs. Figure 5-7 and Figure 5-8 illustrate the exception mechanism with a code example. Figure 5-7 shows how a method is declared normally. No possible exceptions are declared, so the caller can expect the method to execute correctly. Figure 5-8 shows a method that declares that it may throw a `DuplicateNameException`. The method checks some conditions that must be met for the

method to be able to execute correctly (in this case it would check whether the name supplied to the method is not already in use), and throws a `DuplicateNameException` if the conditions are not met. Otherwise, it executes normally.

```
public void createEntity(String name) {  
    // create the entity  
    entity = new Entity(name)  
}
```

Figure 5-7: Declaration of a normal method.

```
public void createEntity(String name) throws DuplicateNameException {  
    // check some conditions  
    ....  
    if (conditionsFailed) {  
        // throw the exception  
        throw new DuplicateNameException();  
    }  
    else {  
        // create the entity  
        entity = new Entity(name)  
    }  
}
```

Figure 5-8: Declaration of a method that throws an exception.

Exceptions thrown by methods must be ‘caught’ by the code that called the method. The exception contains a message, conveying information about why the exception occurred. The user interface module can handle exceptions by displaying the exception’s message on screen. This mechanism provides GKOM with the functionality it needs to offer modeling support. By throwing an exception, GKOM can abort actions that would make a model invalid and explain to the user why the suggested action cannot be performed.

Exceptions are like any other object in Java; they can have attributes and functions. This means that an exception need not be only a message. GKOM has three different kinds of exceptions to cover the support described in this chapter: one for when duplicate names are used, one for when a modeler creates an occurrence in a way that is not permitted by the type-level and one for when a proposed change would make the model inconsistent. Each of these exceptions carries information that the user interface module can use to provide better feedback to the user. The next three sections describe the three exceptions and the information they provide.

Naming errors: the `DuplicateNameException`

As explained in section 5.2 duplicate names always occur in a particular context and only occur when two model ingredients of the same type contend for the same name. The context can be a model, a scenario or model fragment or a quantity space. A `DuplicateNameException` has three fields:

- The name that the modeler suggested.
- The kind of model ingredient that the modeler was creating or renaming.
- The model ingredient that vetoed the use of the name. This would be a model, scenario or model fragment or a quantity space

In addition to these fields, the exception contains a default message, such as ‘an entity called *container* already exists in this model’. The user interface module can simply display the default message on the screen, but it could also create a message that is formatted for easier reading (put the suggested name in boldface, for example) or create a message in a different language than English.

Conflicts between the type and occurrence-level: the `InvalidOccurrenceException`

Whenever a modeler creates or alters a model ingredient at the occurrence-level, GKOM checks whether these model ingredients adhere to the constraints at the type-level. If this is not the case, an `InvalidOccurrence` is thrown. This can happen in a large number of situations and, consequentially, the `InconsistentModelException` is the most versatile of the three exceptions described here. What members it has depends on the context in which the exception is thrown, and is described in detail in the GKOM API documentation. Developers may use these members to offer better support, but may

also choose to simply display the exception message, which gives an accurate description of the problem in each of the contexts.

Changes that lead to inconsistencies: the `InconsistentModelException`

Inconsistencies in a model would arise when a change in one model ingredient causes dependent model ingredients to become inconsistent. Changing model ingredients at the type-level can cause inconsistencies to their occurrences. Changing the type of an entity can make properties, relations and quantities of that entity invalid. This kind of error is the result of modifying knowledge. It occurs when modifying knowledge leads to a configuration that could never have been built from the ground up, because `InvalidOccurrenceExceptions` would have been thrown.

Besides the default exception message, an `InconsistentModelException` contains a set of model ingredients that would become invalid by the suggested modification. When the modeler would try to remove an entity type from the entity hierarchy, for example, the `InconsistentModelException` thrown as a result would contain all occurrences of that entity type. The user interface module can present that set to the user, who can have a look at each of the model ingredients in the set to determine how to resolve the problem.

5.6 Concluding remarks

Modeling support is an essential part of a modeling environment, and even more so for educational software. Incorporating it into the GKOM module increases its utility and makes it easier for application developers to create robust and articulate modeling applications. That said, the support offered by GKOM focuses on correct use of the knowledge articulation language, and application developers may want to extend it with their own, extended support features.

The next three chapters offer a detailed, technical description of how the GKOM module is implemented, and is intended for application developers building modeling applications based on the GKOM module.

6 GKOM fundamentals

This chapter and chapters 7 and 8 describe the implementation of the GKOM module in detail. In chapter 4, four sub-modules of GKOM were identified: the API module, the knowledge-capture module, the import-export module, and the communication module. This chapter and chapters 7 and 8 focus on the API and knowledge-capture modules. The import-export and communication modules are conceptually less interesting and are only briefly described.

One can only conceptually discriminate between the API module and knowledge-capture module. As in any object-oriented design, the classes and their methods that make up the knowledge-capture module *are* the Application Program Interface. This chapter and the next two take a programmer's view; describing the classes that make up the knowledge representation and the operations these classes perform¹. In brief: the knowledge representation is described by its API.

This first chapter describes the foundation of the knowledge representation: the first two levels of the GKOM class hierarchy. This includes the abstract classes Type, Occurrence, and Instance; the base classes for the three knowledge levels described in section 4.3.2. It also includes classes Model, Simulation, State, and Transition, which fall outside the knowledge levels. Chapter 7 describes the classes representing the 'building blocks' of the GKOM knowledge representation: entities, properties, relations, quantities, quantity spaces, values, derivatives and dependencies. Chapter 8 introduces the model fragment and scenario classes.

6.1 Overview

The GKOM module consists of three separate packages²: `gkom.model`, `gkom.parsers`, and `gkom.server`. The classes in each of these packages serve the following purposes:

- **`gkom.model`**
Contains the classes that make up the knowledge representation.
- **`gkom.parsers`**
Contains classes that can transform a GKOM model into an external representation and vice versa. These classes make up the import-export module of the GKOM module.
- **`gkom.server`**
Contains classes that handle the communication with the remote simulation module. The classes in this package make up the communication module.

The primary focus of this chapter lies on the classes found in the `gkom.model` package and the functionality they offer.

Language conventions and the use of UML

Two domains meet in this chapter: that of Qualitative Reasoning and that of the Object Oriented Paradigm. Unfortunately, a number of terms are used in both domains, but have a slightly different meaning in each of them. First of all, a class in OOP is often referred to as a *type*. In GKOM, Type refers to either the abstract class Type or to one of its child-classes. Similarly, the term *instance* in the OO domain is used to refer to objects of a particular class, while in GKOM the term refers to the Instance class or one of its child-classes. In this chapter, the term 'type' is exclusively used to refer to the Type class or its descendants and the term Instance is exclusively used to refer to the Instance class and its descendants. Types and instances in the OO domain are called classes and objects, respectively.

The term 'property' is also used in both the GARP domain and the OO domain. In GKOM, 'property' is the name of one of the modeling primitives. In the OO paradigm a property is a member of a class that can either be edited directly, by assigning a value to it, or indirectly, by calling the

¹ Developers should consider chapters 6, 7 and 8 a comprehensive introduction to GKOM, but a far more detailed description is the GKOM API documentation. It documents all classes and the methods they consist of and is the hands-on guide for developers implementing GKOM. At the time of writing we expect to make the API documentation available online at <http://www.swi.psy.uva.nl/projects/GARP/index.html>.

² A package in Java is similar to a directory in a file system. It contains a set of related classes with special access privileges to each other's methods and attributes.

appropriate get and set methods. To distinguish between a property in OOP and a property in GARP, properties in the OO domain will be referred to as attributes (as is common in UML) or members, and the term ‘property’ is only used for the corresponding GKOM primitive.

Another possible source of confusion is the fact that GKOM uses up to three classes to represent a single GARP primitive: a child-class of Type, a child-class of Occurrence and a child-class of Instance. For example, the classes QuantityType, Quantity, and QuantityInstance all represent the GARP primitive quantity. To discriminate between a statement made about the primitive in general (‘an quantity has a name’, for example) or about one of the particular classes that represent that primitive (‘an QuantityType has a name which is unique in the context of its Model’), the classes are always referred to by their Java class name (EntityType, Entity, EntityInstance) and the primitives in general are written without capitals.

Each of the classes described in this chapter is depicted by a diagram. These diagrams are drawn according to the UML standard for class diagrams (Fowler, 2000), but deviate from that standard in three ways:

1. The diagrams show operations first and attributes last.
2. Arguments to operations are not shown in the diagram, but explained in the text.
3. When a get and set operation exists for a single attribute, only the attribute is shown, not the two methods³.

The diagrams are often simplified representations of the class they depict⁴. In particular, operations that can only be called from within the class (‘private’ operations) are never shown and often when multiple operations exists that perform the same function on different attributes of a class (such as EntityType’s getProperties, getRelations and getQuantities methods) the diagram will show one general method (in this case getTypes) that represents all of these operations. This is also made clear in the text.

A lot of the classes described in this chapter have operations that return sets of objects. In GKOM, sets are always returned as arrays. To show that a method returns an array, the return type in the diagram is suffixed with ‘[]’. This is the token by which Java identifies arrays.

6.2 Top-level classes

Figure 6-1 shows levels 0 and 1 of the gkom.model class hierarchy. All the classes in the gkom.model package inherit from a single parent: class KnowledgeObject. The first three children in the figure are the base classes for the three knowledge levels Type, Occurrence and Instance. The other four children of KnowledgeObject, Model, Simulation, State and Transition, represent GARP primitives that cannot be considered a type, occurrence or instance. In the following sections each of the classes in Figure 6-1 will be described in detail.

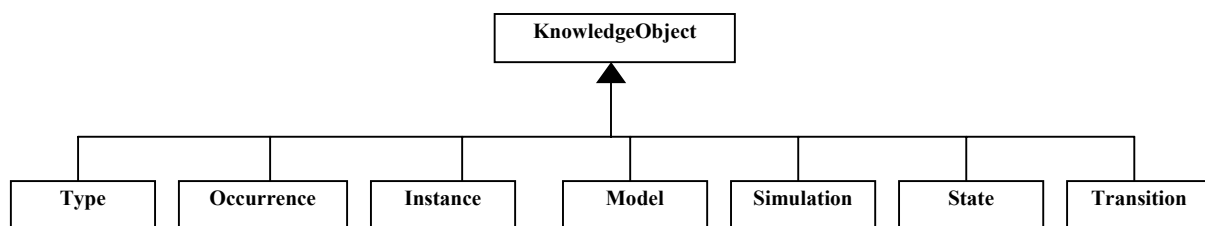


Figure 6-1: levels 0 and 1 of the gkom.model package class hierarchy.

6.2.1 The KnowledgeObject class

All the classes in the gkom.model package that represent an item in the knowledge representation are children of class KnowledgeObject. KnowledgeObject is an abstract class; it cannot exist by itself. Its

³ In Java it is considered good programming practice to have a class hide its actual attributes, allowing access to them only through a get and set method. Because of this, a pair of get and set methods is often also referred to as an attribute.

⁴ The full description of all operations and attributes of the classes in GKOM can be found in the GKOM API documentation in the appendix.

purpose is to offer functionality to all of its children and to simply be the root of the hierarchy so other parts of the module can distinguish between knowledge objects and other types of objects.

KnowledgeObject
isSane: boolean getInsanities: String[] destroy
name: String description: String

Diagram 6-1: Class KnowledgeObject.

A KnowledgeObject has two attributes: a name and a description. There are methods to get and set the description and there is a method to get the name, but there is no method to set the name. This is because not every child-class of KnowledgeObject can have a name set: a Property, for example, is simply named after its PropertyType, as will become clear in the section about properties. Implementing a setName method is left up to the child-classes.

KnowledgeObject as an observable

KnowledgeObject is a child-class of `java.util.Observable` rather than of `java.lang.Object`. The Observable class offers a basic implementation of the Observer pattern (Gamma, 1995), which allows an *observable* to notify *observers* of a change in its state without knowing anything about the observers. With this pattern an application *observing* elements in a GKOM model is notified of any changes in the elements observed. For example, a GUI element displaying an Entity can register as an observer of the Entity it displays so that it will be notified when the name of the EntityType the entity belongs to (or simply its own name) changes. It can then update the display accordingly.

Class Observable provides methods to add observers to it and remove observers from it and offers a method to notify all observers of changes. Any object of a child-class of KnowledgeObject send its observers a notification whenever a change occurs that could be of interest to the application using the GKOM library. The exact events that trigger a notification fall outside the scope of this chapter, but the GKOM API documentation describes them in detail. Two methods in the KnowledgeObject class cause observers to be notified: the setName and destroy methods. All direct and indirect children of the KnowledgeObject class inherit this functionality.

Inspection through isSane and getInsanities

KnowledgeObject also offers a simple mechanism to help its child-classes in performing sanity checks. Child-classes overwrite the isSane method to perform a self-check, determining whether an object of the class is in a valid state. The contract that KnowledgeObject has with its child-classes is that the isSane call should return either true or false and that any deviations from a sane state (that is, any insanities) should be returned by a call to getInsanities. This method returns a list of messages that the object doing the sanity check has filled with errors, if any.

Every direct or indirect child of KnowledgeObject implements the isSane method. KnowledgeObject itself maintains the array of messages that child-classes can write to. This simple mechanism will most likely be more of a help to a programmer implementing GKOM in a modeling environment than to a user building a model. For the programmer it can be an effective tool.

6.2.2 The Type class

The Type class is parent to all classes that represent GARP primitives at the type-level. It is an abstract class; it is only used as the parent-class of all other classes of the Type-level. It offers some functionality common to all Types to its children.

The primary functionality that class Type offers to its child-classes is keeping track of a set of occurrences and instances. As will become clear in the sections about the Occurrence class (Section

6.2.3), every occurrence has a reference to its type⁵. It will also be shown that an occurrence needs this information to be able to offer support.

The reverse is also true. In order for a type to determine whether a change made to it will effect any of its occurrences, it needs to have references to all of its occurrences. The `addOccurrence`, `removeOccurrence` and `getOccurrences` methods of the `Type` class offer a framework for storing this information. When an occurrence is created, it calls its type's `addOccurrence` method and passes it a reference to itself. When the occurrence is destroyed it calls its type's `removeOccurrence` method. Both methods are defined in the `Type` class. The resulting set of all occurrences added and not yet destroyed is returned by method `getOccurrences`.

The `addInstance`, `removeInstance` and `getInstances` methods defined in class `Type` offer the same functionality as described above for a type's instances. There is no strict need for this information from a GKOM perspective (it is not needed for offering support), but it may be of interest to the modeler and achieves consistency in the GKOM API.

All child-classes of `Type` overwrite the `getOccurrences` and `getInstances` methods to return a set of objects of the proper class. An `EntityType`, for example, needs to return a set of `Entity` objects when its `getOccurrences` method is called and a set of `EntityInstance` objects when its `getInstances` method is called. These methods will not be shown in the diagrams of the individual child-classes of `Type`, since their functionality is not clear.

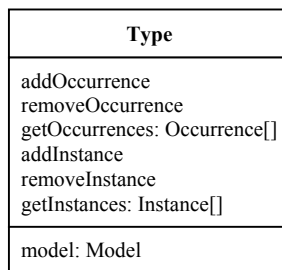


Diagram 6-2: Class Type.

A type is always created in the context of a `Model`. In fact, as outlined in section 6.2.5, the `Model` class builds objects of child-classes of `Type` and sets their `model`-attribute. Types need this attribute for offering support: whenever an operation is called on a type that could make it invalid in the context of the `Model` it is part of, the type will consult its `Model`. An obvious example is when a type's name is changed: it will need to ask its `Model` whether the new name is not already in use.

6.2.3 The Occurrence class

Class `occurrence` is the parent class for all classes that represent a GARP primitive at the occurrence-level. Like class `Type` it is an abstract class: it offers functionality to its children but cannot be instantiated.

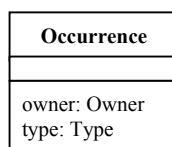


Diagram 6-3: Class Occurrence.

All occurrences have two things in common: they are of a particular type and are part of either a model fragment or scenario. Class `Occurrences` holds a reference to both: its `type`- and `owner`-attribute. Both are used in offering support.

An occurrence's type defines its valid use. When an `Entity` is given a `Property`, for example, it needs to verify whether its type (an `EntityType`) allows it to have the `Property`. The `type` attribute is an attribute of the `Occurrence` class that children of `Occurrence` inherit. Children do not edit the attribute directly, but instead use `Occurrence`'s `setType` and `getType` methods. The `setType` method also takes

⁵ An occurrence is an object of a child-class of the `Occurrence` class and a type an object of a child-class of the `Type` class.

care of calling the `addOccurrence` method of the type given, thus allowing a type to keep track of its occurrences through the framework described in section 6.2.2.

All child-classes of `Occurrence` overwrite the `getType` method to return an object of the proper class. A `Relation`, for example, will return a `RelationType` object rather than a `Type` object.

The owner attribute of class `Occurrence` is a reference to the construct that an occurrence is part of. An occurrence is always part of a construct in the same way that a type is always part of a `Model`. There are two constructs: `ModelFragmentType`, and `ScenarioType`. The constructs all implement the `Owner` interface, so that an occurrence can remain unaware of whether it belongs to a `ModelFragmentType`, `ScenarioType` or `RuleType`, but can simply assume to belong to an `Owner`. The attribute is used in support: when a modeler changes the name of an occurrence (only `Entities` and `Quantities` have a name that differs from their type's name), the occurrence verifies the uniqueness of that name in the construct it is part of by asking its owner whether the new name is in use. Sections 7.1.2 and 7.5.2 describe this form of support in more detail.

6.2.4 The Instance class

The instance class is the last of the classes representing one of the three knowledge levels described in chapter 4. Like `Type` and `Occurrence`, `Instance` is an abstract class that only serves to offer functionality to its child-classes.

There are three fundamental differences between occurrences and instances:

- Instances hold different information in different states. In a simulation, an instance will most likely exist in several states. But the instance's behavior will likely be different in each state. A quantity will have different values in different states, for example.
- Instances are not created by the modeler. The modeler creates the generic part of a model: the types and occurrences. By creating a scenario, the modeler specifies the starting-point of a simulation. Instances are the result of that simulation. The simulation module (which is not part of the GKOM module, see chapter 4) runs a simulation, sends the result of that simulation to the communication module, which hands the results to the parser, which creates the actual instances.
- Instances have no built-in support. Because instances are not created by the modeler, there is no need to offer support. The simulator bases the simulation on the input from the modeler, and the modeler receives support while building the model.

Because all instances belong to one or more states, the obvious place for defining functionality to keep track of the states an instance belongs to, is in the `Instance` class. When an instance is added to or removed from a `State`, that `State` will call the instance's `add-` or `removeState` methods. The instance keeps track of the states added and not removed, allowing a user to ask an instance in which states it exists by calling the `getStates` method. The `existsInState` method is a convenience method that takes a state number and returns whether or not the instance exists in that state.

Instance
<code>addState</code> <code>removeState</code> <code>getStates: int[]</code> <code>existsInState: boolean</code>
<code>type: Type</code>

Diagram 6-4: Class Instance.

The `type` attribute of class `Instance` works in the exact same way that the `type` attribute of class `Occurrence` does. As the `setType` method of `Occurrence` calls the type's `addOccurrence` method, so the `setType` method of `Instance` calls the type's `addInstance` method. All children of the `Instance` class overwrite `Instance`'s `getType` method to return an object of the proper class.

6.2.5 The Model class

Objects of the Model class represent, as the name implies, a complete GKOM model. Class Model is the starting point for every modeling enterprise. An object of the Model class has two important roles in the modeling process:

- It acts as a placeholder for all elements that are part of the model. None of the other modeling primitives can exist outside the context of a Model.
- It acts as a builder Types, which make up the model's ontology.

The first role is that of a container: all elements created in the process of building a model are in some way a member of a Model object. The architecture of the Model class is such that:

- All types created in a modeling process are direct members of a Model object. A model provides access to these types through methods that return an individual type by a specified name, or a list of all types of a certain class.
- All occurrences created in a modeling process are members of ScenarioTypes (see section 8.1) or ModelFragmentTypes (see section 8.2), which, being types, are themselves members of a particular Model object.
- All Instances created by the simulator based on the knowledge in a model are members of States and Transitions (see sections 6.2.6.2 and 6.2.6.3), which themselves are members of Simulation objects (see section 6.2.6), which are members of a Model object.

This architecture makes GKOM different from GARP in two ways. First, a modeling application using GKOM may allow a user to work on several models concurrently. Every model is represented by an object of the Model class, and there is no restriction on how many objects of a class can be created. Application builders can create modeling applications with GKOM that allow the user to switch between models in the same way that a text editor allows users to switch between files. This is not possible in GARP, which can only run one model at a time.

Second, GKOM stores the result of simulations in the Model object, allowing users to revisit these results at some later time. Simulations are represented by objects of the Simulation class, which are members of a Model object. The Model object files Simulations under the ScenarioType they are based on, which means that a Model can store one Simulation per scenario. Simulations remain members of a Model until the user removes them.

The second role of the model class is that of a builder for its own members. This requires a short explanation. For a long time in the GKOM development process, all elements in the `gkom.model` package had public constructors⁶. But when the work began on ensuring the consistency of a model and on offering support to the user, it became apparent that this approach would allow a user too much freedom. To effectively ensure consistency and offer support, the library must assume that certain attributes of objects in a model will be set at creation time and do not change afterwards. For types, the problem is limited to their Model-attribute. A type's Model must be set at construction time, because the consistency and validity checks a type performs are often dependent on information from the Model. An example of this is when the name of a type whose name must be unique, is changed: it will need to ask its Model whether that name already exists or not. This exemplifies that GKOM must assume the model-attribute to exist in order to check for duplicate names at all. It also shows why the model-attribute should not be allowed to change: Changing it would change the context in which it was decided that the name is valid. Also, it would lead to inconsistencies between the Type itself (which would now effectively be part of a different Model) and the Model it was part of (which still holds a reference to the Type as one of its members and which might still contain other members that refer to the type removed).

To prevent these problems from occurring, Model was designed to be in control of creating and destroying objects of the child-classes of Type and of setting the Model property of these types. Model's create methods ensure the proper initialization of types and add the newly created types to the Model, and Model's remove methods ensure that no inconsistencies in the model result from removing a type. Model is not the only class that uses this special-purpose implementation of the Builder pattern

⁶ In Java, the constructor is the method that creates a new object of a particular class. Having a public constructor means that an application using the class can create new objects at will. When a constructor is protected, new objects can only be created by objects of other classes in the same package. This allows the creator of the package tighter control over how objects are created.

(Gamma et al, 1995). ScenarioType and ModelFragmentType handle creation and removal of their members in the same way.

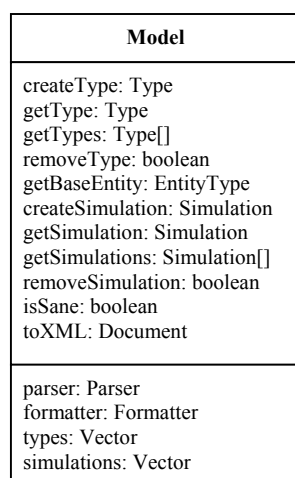


Diagram 6-5: Class Model.

Diagram 6-5 shows a simplified UML class diagram of the Model class. It is simplified in that the createType, getType, getTypes and removeType methods do not actually exist; they represent the methods that operate on each of the individual types. These methods are named after the types they operate on. Thus, method createType in the diagram represents methods createEntity, createProperty, createRelation, createQuantity, createQuantitySpace, createModelFragment, and createScenario. The same holds for getType, getTypes and removeType in the diagram.

Each of the createType methods takes a String as argument: the name for the type to create. If the support system does not intervene (see the section about support below) a new type is created, added as a member of the Model and returned.

Each of the getType methods takes the name of a type as argument and returns the corresponding type, if it is a member of the Model. The getTypes methods return all types of a particular kind (getEntities returns all EntityTypes, getRelations returns all RelationTypes, etc). The removeType methods remove a type of a particular kind by name. They return a boolean (a true or false flag) indicating whether the name given matched a Type in the model; in other words whether there was anything by the name given to remove.

The only direct members of the Model class that are not types, are Simulation objects. Simulations are created by calling the createSimulation method, which requires passing it a ScenarioType on which the Simulation will be based. Unlike the createType methods, this method will not be called by the modeling application in response to some user action, but by the import-export module in response to receiving simulation results from the communication module. Calling the getSimulation or getSimulations methods retrieves Simulations from a Model. The former method is given a ScenarioType and returns the Simulation based on it (if it exists) and the latter returns all simulations in the Model. The removeSimulation method removes the Simulation based on the given ScenarioType.

As all child-classes of KnowledgeObject do, Model overwrites the isSane method. It implements it by forwarding the call to each of its members, adding any ‘insanities’ the members report to its own insanities buffer. A call to a Model’s isSane method thus returns the sanity-status of the entire model; Model’s isSane method will return true only if all members return true. A subsequent call to getInsanities will return any error messages reported by any members.

A Model is not created empty, but instead contains one object of the EntityType class by default. This EntityType is the root of the entity type hierarchy and is referred to as the base entity. Any EntityType created by Model that is not given a parent is given the base entity as parent. This way, model enforces that there be a single EntityType at the root of the hierarchy. Such a single root is desirable because it creates an EntityType that is used by PropertyTypes, RelationTypes and QuantityTypes that are valid

for all entities in a Model. Method `getBaseEntity` retrieves a Model's base entity. Removing the base entity is not possible, as will be shown in the following section.

Model's role in support

Class Model plays a central role in the modeling process, and it plays an equally central role in supporting the modeler. The supporting role can be divided in three parts:

Firstly, Model prevents certain modeling errors by design. As explained above, it acts as the builder for the types it contains. Having complete control over the creation process, Model prevents the modeler from:

- Creating a type outside the context of a Model. After a type is constructed it must be added to a Model and its model-attribute must be set accordingly. Before this, it is just an object in memory, without any reference to the model it was created for. A Model object adds newly created types to itself and sets their model-attribute before returning from the create method.
- Creating a type without a name. A model ingredient is not valid before it has a name. All create methods of the Models class take a String as an argument, which is passed to the type created as its name.
- Creating an EntityType outside the entity type hierarchy. An EntityType is not part of the entity type hierarchy until it has a parent. Models' `createEntity` method requires a parent EntityType as an argument, allowing the method to place the new EntityType in the hierarchy. If a null reference is passed, the Model's base entity is set as the parent.

The second part of Model's role in support is preventing the modeler from using the same name for two different objects. Objects of all child-classes of must have unique names in GKOM. This restriction applies for each child-class of Type individually; there is no restriction between different child-classes of Type⁷. A Model object makes sure unique names are used by throwing a `DuplicateNameException` when a modeler tries to create a new Type with a name that matches the name of an existing type of the same class.

Model's check at creation time is only one part of ensuring that names are kept unique in a Model. Each individual type also checks for duplicate names when its name is changed.

The last part of Model's supporting role is monitoring the consequences of actions performed by the modeler, preventing actions that would make the Model's state inconsistent. Model does this by throwing an `InconsistentModelException` when an action is suggested which would introduce inconsistencies. In general, `InconsistentModelExceptions` are thrown whenever a model ingredient that is referred to by other model ingredients is removed from its owner. In the context of a Model, this can happen in two situations:

- When a type that has occurrences inside `ScenarioTypes` or `ModelFragmentTypes` is removed from a Model. Because all occurrences have a reference to their type, once occurrences of a type have been created, that type should not be removed from the Model.
- When an EntityType that has `PropertyTypes` or `QuantityTypes` or acts as left or right hand side of `RelationTypes` is removed from a Model. `PropertyTypes` and `QuantityTypes` hold a reference to the EntityType they belong to and `RelationTypes` hold a reference to their left and right hand side EntityTypes. Once an EntityType is referred to in such a way, that EntityType can no longer be removed from the Model.

This does not imply that, once a type has been referenced by other model ingredients in some way, it can no longer be removed. It means that inconsistencies that would result from removing the type must be solved before removal. As an aid to the user, a Model object will add references to all inconsistent elements to the `InconsistentModelException` that it throws. The user can use this to solve the inconsistencies and then suggest the action again.

6.2.6 Simulations

There are four classes at the first level of the GKOM class hierarchy that aid in the representation of simulations run by the simulation module: the Instance class and the classes Simulation, State and Transition. The Instance class is parent to the classes representing individual modeling primitives at

⁷ As an example: no two EntityTypes may have the same name, but an EntityType can theoretically have the same name as a RelationType.

the instance-level. State and Transition objects, which represent the corresponding GARP concepts, contain the individual instances. Simulation objects encapsulate State and Transition objects belonging to a single simulation and have a reference to the ScenarioType they were based on. Figure 6-2 illustrates how simulation results are organized in GKOM.

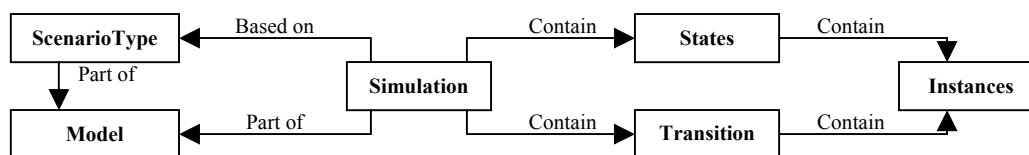


Figure 6-2: Simulations in GKOM

The Simulation class has two important roles in representing simulator output. First, it serves as a container for the State and Transition objects resulting from a simulation based on a single scenario. This makes it possible to store several Simulation objects in one Model object, each representing the results of a simulation based on a different scenario. Second, it serves as a container for the individual instances in a simulation. This facilitates representing an instance appearing in multiple states and transitions as a single object. This approach is different from the one taken by GARP. As Figure 6-3 illustrates, GARP instances that appear in several states are declared in each state that they appear in.



Figure 6-3: GARP organisation of instances and states.

In GKOM, all instances in a simulation are defined in (and constructed by, see the next section) a Simulation object. The State objects (themselves part of the Simulation object) contain references to them.

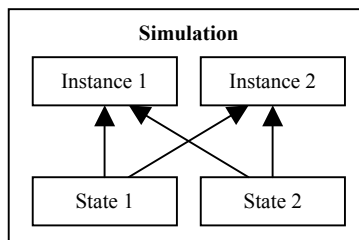


Figure 6-4: GKOM organisation of instances and states.

The GKOM approach has a number of advantages. First of all, Instances that exist in several states are only created once, saving processing time and memory overhead. Second, a Simulation object provides easy access to all instances that exist in it, where in GARP retrieving that information will require going over all states to find all unique instances. Third, individual instances are knowledgeable of the State objects that reference them; they know in which states they exist.

The next three sections describe the Simulation, State and Transition classes in more detail.

6.2.6.1 The Simulation class

Object of the Simulation class represent the result of a simulation run by the simulator based on a specific ScenarioType. They build and contain all individual instances and the states and transitions that make up a simulation. They are members of a Model object, which can hold one Simulation for every ScenarioType it contains.

Simulation
createInstance: Instance getInstance: Instance getInstances: Instance[] removeInstance createState: State getState: State getStates: State[] createTransition: Transition getTransitions: Transition[] removeTransitions setScenario: ScenarioType
model:Model

Diagram 6-6: Class Simulation.

Like Model object, Simulation objects serve as the builder for their own members. This builder-functionality is used by the import-export module. When that module receives new simulator results from the communication module, it creates a Simulation object and uses that to build States, Transitions and individual instances for the elements that it finds in the simulator output. It uses the createInstance methods of class Simulation to do so. The Simulation object keeps track of which instances it has created and makes sure that every unique instance is created only once.

Diagram 6-6 shows Simulation to have methods createInstance, getInstance, getInstances and removeInstance, but these are not the actual method names. Simulation has create, get and remove methods for each of the individual instances⁸. The create methods must be passed all the information needed to uniquely identify the instance that is to be created. For some instances this is simple; EntityInstances and QuantityInstances, for example, have names that are unique throughout a simulation. The createEntity and createQuantity methods are passed an EntityType or QuantityType, respectively, and a name. The name is enough for Simulation to find an existing EntityInstance or QuantityInstance if previously created. Other create methods must be passed additional arguments. A RelationInstance, for example, can be uniquely identified by a combination of its Type and its left and right hand side EntityInstances.

The getInstance methods return individual Instances of a particular class and the getInstances methods return all instances of a particular class. These methods will find Instances in all of the Simulation's States and Transitions. The removeInstance methods remove Instances from the Simulation. Like with the create methods, users should not call these methods directly. Information at the instance-level is the result of the simulator's work and should not be changed by any other object than a parser.

Simulation also creates its States and Transitions. States are created by the createState method, which must be given a state number. Method getState returns the state with the given number and method getStates returns all states in the simulation. There is no method to remove a state, because states never 'disappear' in the course of a simulation. Transitions can be removed (by calling removeTransitions), but cannot be retrieved individually, because they can't be uniquely identified. Method getTransitions returns all the Transition objects in a Simulation.

The model property of the Simulation class holds a reference to the Model the simulation is part of. It needs this property in only one situation: when a Simulation is destroyed it tells its Model to remove it from the Model's list of Simulations.

A Simulation does not have a name. But as a child-class of KnowledgeObject, which has a getName method, it must have an answer when asked for one. In that situation it returns the String 'simulation_of_[name]', where [name] is the name of the ScenarioType that the Simulation is based on.

⁸ The complete list is EntityInstance, PropertyInstance, RelationInstance, QuantityInstance, ValueInstance, DerivativeInstance, InequalityInstance, CorrespondenceInstance, CausalDependencyInstance, ModelFragmentInstance, TerminationRuleInstance, PrecedenceRuleInstance, ContinuityRuleInstance and ScenarioInstance.

6.2.6.2 The State class

A State object represents a state of behavior inside a Simulation. Its owner is a Simulation object, which will usually contain several states. Every State holds a ScenarioInstance object representing the current state of the ScenarioType on which the Simulation was based. In addition it holds all Instances active in the state. A state number identifies it.

State
addInstance getInstance: Instance getInstances: Instance[] removeInstance: boolean
stateNr: int status: int scenario: ScenarioInstance simulation: Simulation

Diagram 6-7: Class State.

State implements add, get and remove methods for all Instances except RuleInstances and ScenarioInstances. RuleInstances are not part of states and a State's current scenario is a property rather than a list. As with the Simulation class described above, the add and remove methods should only be called by parsers. A modeling application will only find the get methods useful.

The status property can be set to one of *interpreted*, *terminated*, *ordered* or *closed*. This is the only property of a State that changes during the simulation; apart from it a State remains the way it was created. The scenario property is, as mentioned above, a ScenarioInstance object representing the current state of the initial scenario. The rather unique characteristics of the ScenarioInstance object are described in the section about the scenario. State holds a reference to the Simulation it is part of in its simulation property.

When asked for a name, State answers with 'state[n]', where [n] is the state number.

6.2.6.3 The Transition class

The last of the seven children of KnowledgeObject is the Transition class. Transitions are part of a Simulation and hold knowledge about how a simulated system moves from one state into the next.

Transition
getConditions: Transition.Conditions getGivens: Transition.Givens createRule: RuleInstance getRules: RuleInstance[] addToState getToStates: int[]
state: int status: int simulation: Simulation

Diagram 6-8: Class Transition.

Class Transition has two inner classes, Transition.Conditions and Transition.Givens, which represent the Transition's conditions and givens. The Conditions block contains state-information that caused the Transition to occur, while the Givens block shows how the state-information changed as a result of the transition. They are retrieved by calling getConditions and getGivens, respectively.

A Transition is the result of rules firing based on the state of a simulation. The Transition object is the builder for these rules: method createRule takes a RuleType as an argument and creates a RuleInstance, which it adds to the Transition and returns to the caller. Like the create-methods in Simulation, only the parser is supposed to call method createRule. A modeling application should only be interested in reading the set of rules that caused a Transition. Calling getRules retrieves this set.

A Transition moves a simulation from one state, represented by Transition's state property, to zero or more other states, represented by Transition's toState set. This set is edited through the addToState and getToStates methods.

Apart from its state property, Transition has a status property and a simulation property. The latter is simply a reference to the Simulation the Transition belongs to. The former represents the status of a Transition and can have the same values as State's status property: interpreted, terminated, ordered or closed.

Both the Conditions and Givens classes are simple containers of instances. The instances they hold are managed through the add, get, and remove methods. As before, the methods in the diagram are not the actual ones; there are add, get and remove methods for each of the Instances that a Conditions or Givens can contain. Both Conditions and Givens can contain EntityInstances, PropertyInstances, RelationInstances, QuantityInstances, ValueInstances, DerivativeInstances and DependencyInstances. In addition, a Transition.Conditions object can contain Transition.QSConstraint objects. These represent quantity space constraints at the instance-level. Because quantity space constraints at the instance-level are only found inside Transitions⁹, QSConstraint was made an inner class of Transition rather than a separate class.

Transition.Conditions/Givens
addInstance getInstance: Instance getInstances: Instance[] removeInstance: boolean

Diagram 6-9: Inner classes Transition.Conditions and Transition.Givens.

A quantity space constraint can have one or two values. In the first case it simply serves to state that a quantity can have a certain value of a certain type. In the second case it represents (apart from the two value's types) that one value comes before the other. GARP uses three separate predicates to represent quantity space constraints: the interval, point and meets predicates. The QSConstraint represents a point predicate if it's single value is a point, an interval predicate if it's single value is an interval and a meets predicate if it contains two values.

Transition.QSConstraint
getLowerValue: ValueInstance getUpperValue: ValueInstance getQuantity: QuantityInstance

Diagram 6-10: Inner class Transition.QSConstraint.

QSConstraint objects are initialized with a lower and upper ValueInstance. Methods getLowerValue and getUpperValue, respectively, retrieve them. Method getQuantity returns the QuantityInstance that the ValueInstance or ValueInstances part of the constraint belong to.

6.2.7 Overview of the top-level classes

Sections 6.2.1 through 6.2.6.3 have introduced those classes of the gkom.model package that make up the first two levels of the class hierarchy. The KnowledgeObject class is at the top of the hierarchy; all other classes in the package inherit from it either directly or indirectly. Classes Type, Occurrence and Instance are the parent-classes for the classes that make up the type, occurrence and instance-level of the GKOM model, respectively. The other four classes described (Model, Simulation, State and

⁹ Two types of quantity space constraints are found in the gkom.model package: one inside TerminationRuleTypes and one inside Transitions. The difference between the two is only that the one found in TerminationRuleTypes holds two Value objects, while the one found in Transitions holds two ValueInstance objects.

Transition) are direct children of KnowledgeObject because they cannot be considered a type, occurrence or instance.

Figure 6-5 is a schematic depiction of how the main classes described thus far relate to each other. Not all classes are shown. Class KnowledgeObject is left out because its only relation to all of the other classes is that it is their parent-class. For simplicity, the fact that a Simulation consists of States and Transitions, and that Instances are in fact part of those, is also not shown. Additionally, the figure contains two classes that have not yet been introduced: ScenarioType and ModelFragmentType. They are described in chapter 8

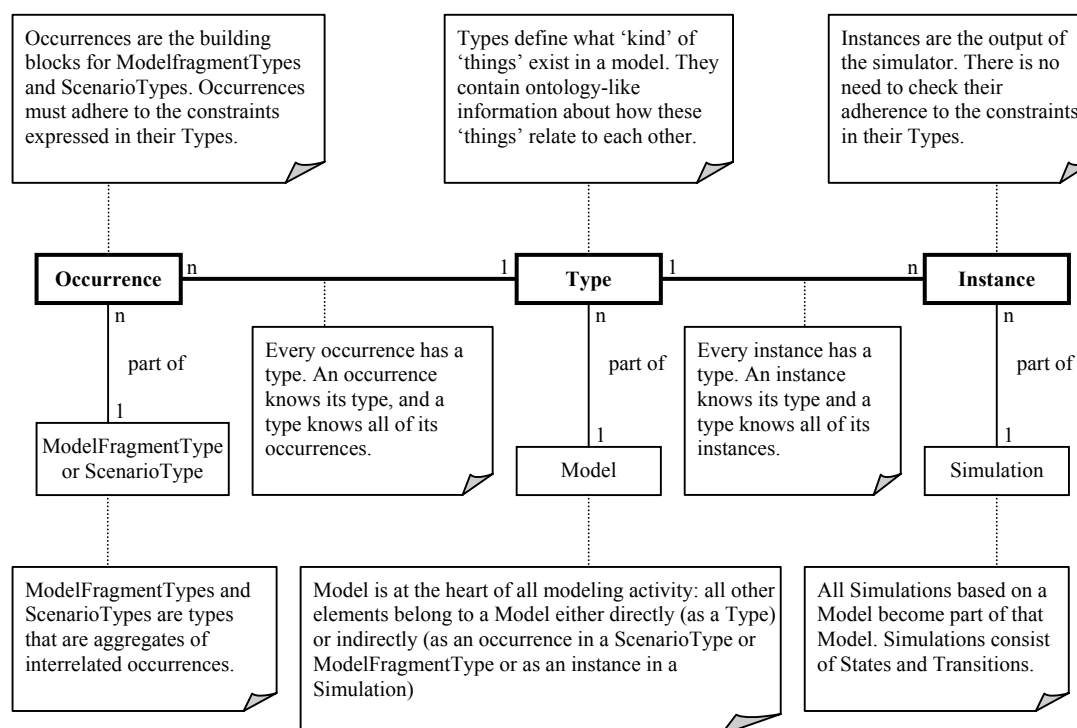


Figure 6-5: An annotated overview of the relations between the top-level classes.

The top part of Figure 6-5 shows how types, occurrences and instances are related. It should be noted that the Type, Occurrence and Instance boxes in the figure do not refer to objects of the corresponding classes, but instead represent objects of any of their child-classes. But what the figure shows holds for objects of all the child-classes of Types, Occurrences and Instances in general: every occurrence and instance has a reference to a type and every type has references to all of its occurrences and instances.

The bottom of the figure shows that types, occurrences and instances are always part of objects of other classes in the GKOM library. In fact: all objects are in some way part of a model object. In any given model, all types in the model belong to the model directly. Occurrences of the types in the model are part of ModelFragmentTypes or ScenarioTypes in the model, which are part of the model directly. And instances of the types in the model are part of a Simulation based on the model, which is also part of it.

The next chapter describes the individual child-classes of Type, Occurrence and Instance.

7 Building blocks

This chapter and the next describe the classes with which a Model is built; the child-classes of the Type, Occurrence and Instance class. This chapter describes the ‘building blocks’. Building blocks are the elements found inside model fragments and scenarios. The next chapter describes Model fragments and scenarios.

This chapter is organized as follows. The main sections 7.1 through 7.11 each discuss one building block. The sub sections of these sections discuss the classes used to represent the building block at the type, occurrence and instance-levels. Every class is introduced with a class diagram and described in terms of its methods and attributes.

7.1 Entities

Entities come in three different forms in GKOM: EntityType as child-class of Type, Entity as child-class of Occurrence and EntityInstance as child-class of Instance. Each plays a slightly different role.

7.1.1 EntityType

Class EntityType represents entities at the type-level. An EntityType is a generic entity in a domain, a class of objects. It is owned by a Model and has a parent and possibly children, placing it in the entity type hierarchy. It serves as a placeholder for information about how its occurrences (Entity objects) should be used inside constructs: by adding PropertyTypes, RelationTypes and QuantityTypes to it, a modeler defines what Properties, Relations and Quantities can be used with occurrences of the EntityType. It has a name that must be unique within the context of a Model’s set of EntityTypes; within one model, no two EntityTypes can have the same name.

EntityType maintains lists of PropertyTypes, RelationTypes and QuantityTypes added to it. These lists offer the modeler a possibility to ‘ask’ an EntityType which PropertyTypes, RelationTypes and QuantityTypes refer to it. Without these lists the modeler would only be able to ask a PropertyType or QuantityType to which EntityType it belongs and ask a RelationType for its left and right hand sides, but would not be able to ask the reverse question (such as ‘give me all PropertyTypes belonging to this EntityType’).

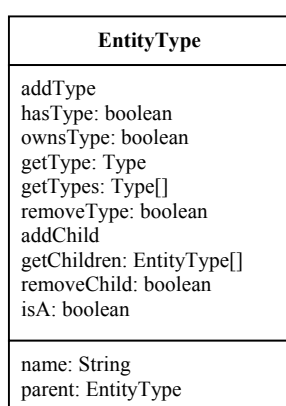


Diagram 7-1: Class EntityType.

With these lists, however, an EntityType is cross-linked with its PropertyTypes, RelationTypes and QuantityTypes. This can only be achieved in a two-step process. To take adding a PropertyType to an EntityType as an example, both EntityType’s addProperty method and PropertyType’s setEntity method need to be called. But it is not desirable to expect the modeler to perform both steps: it is counter-intuitive (the modeler will expect to be finished after the first step) and error prone (forgetting the last step causes inconsistencies in a Model). That is why the addType and removeType methods

shown in the diagram (which add elements to and remove elements from the lists and represent methods `addProperty`, `addRelation` and `addQuantity` and `removeProperty`, `removeRelation` and `RemoveQuantity`) are not accessible outside the `gkom.model` package. In stead `PropertyType`, `RelationType` and `QuantityType` call them, thus establishing the cross-link automatically.

There are `add`, `has`, `owns`, `get` and `remove` methods (the first six methods in the diagram) for `PropertyTypes`, `RelationTypes` and `QuantityTypes`. `EntityType` maintains a list for each of these types. The `getType` methods (`getProperty`, `getRelation` and `getQuantity`) retrieve a reference to an element in one of the lists by name, while the `getTypes` methods (`getProperties`, `getRelations` and `getQuantities`) return all the elements in one of the lists. The `hasType` methods return true when a particular `PropertyType`, `RelationType` or `QuantityType` is an element of the corresponding list or of the list of an `EntityType` up the hierarchy and false if it is not. In other words: an `EntityType` is considered to 'have' not only the `PropertyTypes`, `RelationTypes` and `QuantityTypes` that were added to itself, but also the ones that were added to its parent and all other `EntityTypes` between itself and the Model's base entity in the hierarchy. As a consequence, the `hasType` methods cannot be used to check to which `EntityType` a `PropertyType`, `RelationType` or `QuantityType` was added. To verify whether a type was added to the `EntityType` directly, the `ownsType` methods return true only when the `PropertyType`, `RelationType` or `QuantityType` is part of `EntityType`'s corresponding list.

The `addChild` and `removeChild` methods add and remove children to an `EntityType`'s list of children. Like the `add` and `remove` methods for `PropertyTypes`, `RelationTypes` and `QuantityTypes` they are not accessible outside the `gkom.model` package. Only the `setParent` method is accessible and that method manages adding and removing children to `EntityTypes`. The `getChildren` method returns the list of children and is accessible outside the package. To determine whether an `EntityType` is among the children (either directly or indirectly) of another `EntityType`, the `isA` method can be called. As the name implies, it will return true if the `EntityType` called 'is a' `EntityType` passed to the method.

7.1.2 Entity

Class `Entity` is a child-class of `Occurrence` and represents an entity as used inside a construct: a `ModelFragmentType`, `ScenarioType` or `RuleType`. Through its super class it has a type attribute, pointing to the `EntityType` it is an occurrence of. An `Entity` can have `Properties` and `Quantities` and can be the left or right hand side of `Relations`. These will always be part of the same construct they themselves are part of.

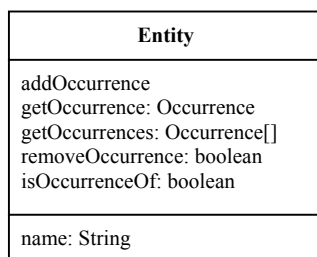


Diagram 7-2: Class Entity.

`Entity` has `add`, `get` and `remove` methods for each of the `Occurrence` types it can refer to: `Property`, `Relation` and `Quantity`. For simplicity, these are shown as `addOccurrence`, `getOccurrence`, `getOccurrences` and `removeOccurrence` in the diagram. `Entity` is cross-linked with its `Properties`, `Relations` and `Quantities` in the same way that `EntityType` is cross-linked with its `PropertyTypes`, `RelationTypes` and `QuantityTypes`, as described above. And like those of `EntityType`, `Entity`'s `add` and `remove` methods can't be called by the modeler, but are instead called by the `Properties`, `Relations` and `Quantities` when their `setEntity` or `setLefthandSide` or `setRighthandSide` methods are called. Only the `get` methods are public, allowing a modeler to ask an `Entity` for all its `Properties` or `Quantities` or all the `Relations` it is part of.

The `isOccurrenceOf` method is a convenience method to determine whether an `Entity` is an occurrence of the specified `EntityType`. The implementation takes into account that the `EntityTypes` in

a Model form a hierarchy and that any Entity that is an occurrence of a particular EntityType, is also an occurrence of that EntityType's parent.

7.1.3 EntityInstance

Class EntityInstance, a direct child-class of Instance, represents entities at the instance-level. They are always part of a Simulation, within which they are part of one or more States. Within the State they can be part of the ScenarioInstance and of one or several ModelFragmentInstances or RuleInstances.

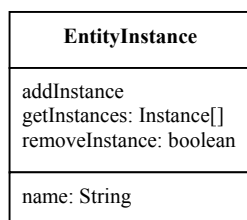


Diagram 7-3: Class EntityInstance.

The EntityInstance diagram closely resembles that of Entity; there are only a few differences between the classes. First, EntityInstance refers to instances of properties, relations and quantities rather than occurrences thereof. Like their counterparts in the Entity class, the add and remove methods are not available outside the package; they are called by PropertyInstance, RelationInstance and QuantityInstance. There are two different kinds of getInstances¹ methods: those that take a state number as argument and those that do not. The former kind return the EntityInstance's set of PropertyInstances, RelationInstances or QuantityInstances active in the state given, the latter kind return the full sets: all PropertyInstances, RelationInstances or QuantityInstances an EntityInstance has.

The name attribute of EntityInstance is like the type attribute of Entity. An EntityInstance has a name that is unique throughout the entire Simulation.

7.2 Properties

A property can be viewed as a named relation between an entity and a fixed value. As discussed in section 4.3.2, GKOM uses two different modeling primitives for the GARP predicate *has_attribute*: *property* and *relation*. Relations are discussed in section 7.3.

Properties exist at each of the three knowledge levels and are represented by the classes PropertyType, Property and PropertyInstance.

7.2.1 PropertyType

A PropertyType is the representation of a property at the type-level and as such, the PropertyType class is a subclass of class Type. A PropertyType has a unique name in the context of the Model it belongs to. It holds two pieces of information: the EntityType it belongs to and a list of possible values. Section 7.2.2 about the Property class explains how this information is used at the occurrence-level.

¹ getInstances in the diagram represents methods getProperties, getRelations and getQuantities in the actual EntityInstance class.

PropertyType
addValue insertValue renameValue getValues: String[] removeValue
entity: EntityType name: String defaultValue: String

Diagram 7-4: Class PropertyType.

The possible values are represented by String objects. Method `getValues` returns the entire list, while methods `addValue` and `removeValue` add values to and remove values from it. Method `insertValue` inserts a value into the list at a given index and method `renameValue` changes an existing value rather than replacing it.

For convenience, a `PropertyType` object can have a default value set. If the modeler sets this value to one of the values in the object's list, every occurrence subsequently created will get that value.

A `PropertyType` is cross-linked with the `EntityType` it belongs to: `PropertyType` has a reference to the `EntityType` and the `EntityType` has a reference to the `PropertyType`. This principle allows the modeler to 'ask' both the `PropertyType` to which entity it belongs and the `EntityType` what `PropertyTypes` it has. Calling `PropertyType`'s `setEntity` method, which is the only method available to the modeler, establishes the cross-link by calling `EntityType`'s `addProperty` method. The latter method is not available to the modeler, as explained in the section about `EntityTypes`.

7.2.2 Property

Class `Property` is a child-class of `Occurrence` and represents properties at the occurrence-level. A `Property` is a member of a `ModelFragmentType` or a `ScenarioType`, which creates it, and belongs to an `Entity`. It is named after its `PropertyType` and can have different values in different contexts.

Property
getName: String setValue getValue: String
entity: Entity

Diagram 7-5: Class Property.

Most of the classes described up to here have had a name-attribute. The `Property` class diagram does not, because it does not have a `setName` method. Properties take on the name of their `PropertyType` and do not have a name of their own. This sets Properties (and, as will become clear below, Relations, Values and Derivatives) apart from Entities and Quantities, which have an occurrence-name apart from their type's name. The only way to change the name of a `Property` is by changing the name of its `PropertyType`, which will also change the name of all other occurrences of that `PropertyType`.

`PropertyType`'s `setType` method cannot be called from outside the `gkom.model` package; it must be set at creation time.

7.2.3 PropertyInstance

At the instance-level, properties are represented by `PropertyInstance` objects. The functionality of the class closely resembles that of class `Property`, the exceptions being that a `PropertyInstance` belongs to an `EntityInstance` rather than to an `Entity` and that a `PropertyInstance` can have different values in different states.

PropertyInstance
getName: String setValue getValue: String
type: PropertyType entity: EntityInstance

Diagram 7-6: Class PropertyInstance

PropertyInstances are part of a Simulation, in which they can additionally be a member of one or more States, Transitions, ScenarioInstances and ModelFragmentInstances. Like Properties, PropertyInstances take on the name of the PropertyType they are an Instance of. The getName method simply forwards the request to the PropertyType. No setName method exists. Both the setValue and the getValue methods take a state number as an argument. Outside the context of a State, a PropertyInstance cannot have a value.

7.3 Relations

Relations are a named link between two entities. The two entities are referred to as the relation's left and right hand side. Classes RelationType, Relation and RelationInstance represent relations at the type, occurrence and instance-levels, respectively.

7.3.1 RelationType

At the type-level, class RelationType represents relations. RelationTypes are members of a Model and their left and right hand side attributes are EntityType objects.

RelationType
getOccurrences: Relation[] getInstances: Relation Instance[]
leftHandSide: EntityType rightHandSide: EntityType name: String

Diagram 7-7: Class RelationType.

Besides getOccurrences and getInstances, which return all of a RelationType's active Occurrences and Instances, a RelationType has get and set methods for its leftHandSide, rightHandSide and name attributes. The left and right hand side EntityType determine how occurrences of a RelationType (Relation objects) can be used, as the section about Relations will show.

7.3.2 Relation

Class Relation forms the occurrence-level representation of a relation. ModelFragmentTypes and ScenarioTypes own and build Relations. The left and right sides of a Relation are Entity objects. Relations do not have a name of their own, instead they are named after their RelationTypes.

Relation
GetName: String
type: RelationType leftandSide:Entity rightHandSide: Entity

Diagram 7-8: Class Relation.

A Relation's getName operation returns the name of the RelationType it is an occurrence of. Setting a Relation's name is not possible, it has to be done by calling the setName operation of its RelationType, which would change the names of all of the RelationType's occurrences.

The type attribute cannot be changed from outside the package (the setType method is protected) and should never change after a Relation has been created.

7.3.3 RelationInstance

RelationInstance is the instance-level representation of a relation. It is part of a Simulation, in which it belongs to one or several States or Transitions. In a State a RelationInstance can be part of the ScenarioInstance and of ModelFragmentInstances.

RelationInstance
GetName: String
type: RelationType leftandSide: EntityInstance rightHandSide: EntityInstance

Diagram 7-9: Class RelationInstance.

The RelationInstance diagram resembles that of Relation very closely. The only difference is that a RelationInstance's left and right hand side are EntityInstance objects, not Entity objects. Like a Relation, a RelationInstance is named after its RelationType and thus has a getName method but no setName method. Also like a Relation, a RelationInstance's type attribute cannot be changed from outside the package.

7.4 Quantity Spaces

Before we can move on to describe the implementation of quantities and values, an explanation is needed about GKOM's representation of quantity spaces. As this section will show, the quantity space is the only building block in the gkom.model package that has only a single class to represent it: class QuantitySpace. It will also become clear that other representations do exist, but are hidden inside Quantity (section 7.5.2) and QuantityInstance (section 7.5.3) objects.

A quantity space defines the range of values that a quantity can have. The values themselves are qualitative states: the flow rate of water flowing trough a pipe, for example, can either be zero (no water flowing trough the pipe) or plus (an unspecified amount of water flowing trough the pipe).

GARP specifies that although two quantities can have the same quantity space, the values of the two quantities cannot be considered equal. This is illustrated in Figure 7-1:

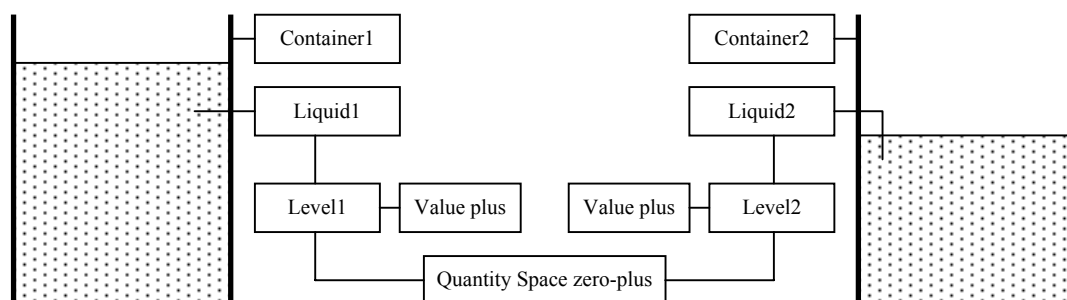


Figure 7-1: Two quantities with the same quantity space and qualitative value. The actual values are unequal.

Quantities Level1 and Level2 both have quantity space 'zero-plus' and the current value of both is 'plus', but value 'plus' of Level1 does not equal value 'plus' of Level2. Translating these requirements to the OO paradigm, GKOM handles quantity spaces and values as follows:

- Objects of the QuantitySpace class represent types of quantity spaces. There is one QuantitySpace object for every unique quantity space. Quantity space zero-plus in Figure 7-1 is represented by a single object of the QuantitySpace class.
- Every Quantity has a reference to its QuantitySpace. Several Quantities can refer to the same QuantitySpace object. Level1 and Level2 in Figure 7-1 both refer to one QuantitySpace object. In short: Level1.QuantitySpace = Level2.QuantitySpace.
- Every Quantity creates its own set of Values. These Values are unique to every Quantity object. In Figure 7-1, Level1.plus ≠ Level2.plus.
- Every QuantitySpace consists of a set of ValueType objects. The Values that a Quantity creates for itself are occurrences of the ValueTypes in its QuantitySpace. In short: Level1.plus.type = Level2.plus.type.

Because every Quantity builds its own set of Values, there is no need for a class that represents quantity spaces at the occurrence-level. In fact: the set of Values a Quantity builds *are* the occurrence-level representation of the quantity space. The same holds for the need to represent quantity spaces at the instance-level: there is no need for it because QuantityInstances create their own set of ValueInstances, thus creating an instance-level representation of a quantity space.

How Quantities and QuantityInstances create their Values and ValueInstances is explained in sections 7.5.2 and 7.5.3. The remainder of this section outlines the implementation of the QuantitySpace class.

Class QuantitySpace is a direct child-class of Type and is thus part of the type-level. It defines the different kinds of quantity spaces in use in a Model. It is owned and created by a Model and is in turn the owner and creator of the ValueTypes (see section 7.6.1) it consists of. It has a name that must be unique: no two QuantitySpaces by the same name can exist in one Model.

QuantitySpace
createValue: ValueType insertValue: ValueType getValues: ValueType[] removeValue: boolean addQuantity getQuantities removeQuantity
name: String model: Model

Diagram 7-10: Class QuantitySpace.

The getValues method of class QuantitySpace returns the actual value range: an array of ValueType objects in a predefined order. The getValue method takes the name of a ValueType and returns the ValueType object, if it exists. QuantitySpace has two operations that add ValueTypes to the set of ValueTypes: createValue and insertValue. Both are given a name and value type (either point or interval; see section 7.6.1), create a new ValueType object accordingly and return that new ValueType object. The createValue method adds the newly created ValueType at the end of the set of ValueTypes; the insertValue method inserts it at a given index. Finally, method removeValue removes the given ValueType object from the QuantitySpace's set.

All types can be asked for all of their occurrences and instances. EntityType's getOccurrences method, for example, returns all Entities of a specific type, as section 7.1.1 explained. But since there is no specific representation for quantity spaces at the occurrence and instance-level, class QuantitySpace does not offer this functionality. However, a QuantitySpace does keep track of all Quantities that use it. Class Quantity has a setQuantitySpace method, which performs a call to QuantitySpace's addQuantity method, passing it a reference to itself. With this information, a QuantitySpace is able to create a list of all Quantities that use it as a QuantitySpace. Method getQuantities returns that list.

7.5 Quantities

Quantities are represented at all three knowledge levels in GKOM: by `QuantityType` at the type-level, by `Quantity` at the occurrence-level and by `QuantityInstance` at the instance-level.

7.5.1 QuantityType

Class `QuantityType` represents quantities at the type-level. `QuantityTypes` have a name and a `QuantitySpace` (see section 7.4) and optionally a list of references to `EntityType`s.

QuantityType
<code>addEntity</code> <code>getEntities: EntityType[]</code> <code>hasEntity: boolean</code> <code>removeEntity: boolean</code>
<code>name: String</code> <code>quantitySpace: QuantitySpace</code>

Diagram 7-11: Class `QuantityType`.

The `addEntity` and `removeEntity` methods edit the list of `EntityType`s a `QuantityType` refers to. The list of `EntityType`s captures information about how Quantities (at the occurrence-level) should be used: any instance of a `QuantityType` can only belong to instances of the `EntityType`s (Entity objects) that the `QuantityType` refers to. The list thus has the same role as the `entity-attribute` of `PropertyType` or the `leftHandSide`- and `rightHandSide`-attributes of `RelationType`. But where a `PropertyType` belongs to a single `EntityType` and a `RelationType` has a single left hand side and a single right hand side, a `QuantityType` can refer to several `EntityType`s in different places in the `EntityType` hierarchy. This allows the modeler a greater degree of flexibility in defining the possible behavior of Quantities: instead of mandating that a `Quantity` belong to an occurrence of one particular `EntityType` or its descendants (as is the case with a `Property`, for example), GKOM allows the modeler to specify a set of `EntityType`s.

Calling the `getEntities` method retrieves the full list of `EntityType`s that a `QuantityType` refers to. Method `hasEntity` is a convenience method that returns whether a particular `EntityType` is part of the list or not.

The `QuantitySpace` attribute serves a double purpose. It allows the modeler to define which `QuantitySpace` is valid for a particular `QuantityType` (or rather: for all the occurrences of the `QuantityType`, see section 7.5.2) and it relieves the modeler of the burden of having to assign a `QuantitySpace` to each of the occurrences of a `QuantityType` individually. Once the `QuantitySpace` attribute of a `QuantityType` is set, all occurrences of it subsequently created will get that `QuantitySpace` automatically.

7.5.2 Quantity

At the occurrence-level, class `Quantity` represents quantities. A `Quantity` has six attributes: a name, a `QuantityType`, an `Entity` to which it belongs, a `QuantitySpace`, and optionally a `Value` and `Derivative`. A `Quantity` has a name of its own: one that can be different from its `QuantityType` name. This is unlike `Properties`, `Relations`, `Values`, and `Derivatives`, which always take on their type's name. The name of a `Quantity` is unique in the `ModelFragmentType` or `ScenarioType` it belongs to.

Quantity
setValue getValue: Value getPossibleValue: Value getPossibleValues: Value[] setDerivative getDerivative: Derivative getPossibleDerivative: Derivative getPossibleDerivatives: Derivative[]
name: String entity: Entity quantitySpace: QuantitySpace

Diagram 7-12: Class Quantity.

As outlined in section 7.4, the quantitySpace attribute of a Quantity is a reference to the same QuantitySpace object that a Quantity's QuantityType refers to. When a Quantity is given a QuantitySpace, it creates occurrences of each of the ValueType objects in it. The Value objects thus created are a Quantity's internal representation of its quantity space and are used as the assigned value of a Quantity (the Value set with method setValue) and in Dependencies (see section 7.8).

A Quantity does not allow manipulation of the Values in its internal representation of its quantity space; no other object than the Quantity itself can change them. To enforce this, the setValue method does not receive a Value object as an argument (which would create the possibility of assigning a Value object that is not in the Quantity's quantity space), but a String object that must be equal to the name of one of the Values. It does, however, allow other objects to query the Value objects it has created. The getPossibleValue and getPossibleValues methods serve this purpose. The getPossibleValue method returns one of a Quantity's Values by name and can be used to obtain a reference to a Value to be passed to a Dependency. The getPossibleValues method returns all of a Quantity's Values and can be used by a modeling environment to display all Values of a Quantity.

A Quantity handles Derivatives in exactly the same way as it does Values. The setDerivative, getDerivative, getPossibleDerivative and getPossibleDerivatives operate on Derivative objects, but perform the same way the setValue, getValue, getPossibleValue and getPossibleValues methods described above do. The only difference between the way a Quantity handles values and the way it handles derivatives, is that the set of possible Derivatives is created immediately when a Quantity is created, whereas the set of Values is created when a Quantity is given a Quantity space. This is because the Derivative quantity space is fixed in GKOM. Section 7.7 explains this.

7.5.3 QuantityInstance

Class QuantityInstance is a direct child of the Instance class and represents quantities at the instance-level. A QuantityInstance has a name that is unique throughout the simulation it is part of, belongs to an EntityInstance, has a QuantitySpace and has a set of assigned ValueInstance and DerivativeInstance objects that can be different in each of the States it belongs to in a Simulation.

QuantityInstance
getValue: ValueInstance getPossibleValue: ValueInstance getPossibleValues: ValueInstance[] getDerivative: DerivativeInstance getPossibleDerivative: DerivativeInstance getPossibleDerivatives: DerivativeInstance[]
name: String entity: EntityInstance quantitySpace: QuantitySpace

Diagram 7-13: Class QuantityInstance.

Like a Quantity, a QuantityInstance creates and manages its own set of possible values (ValueInstance objects) and possible derivatives (DerivativeInstance objects). The getPossibleValues and

getPossibleDerivatives methods return these sets. The getPossibleValue and getPossibleDerivative methods return one value or derivative from the sets.

The getValue and getDerivative methods differ from their counterparts from the Quantity class in that they take a state number as argument. Within a simulation, the values and derivatives of quantities change from state to state and the QuantityInstance object knows what ValueInstance and DerivativeInstance it has in each state. Passing a state number to the getValue and getDerivative methods returns the ValueInstance or DerivativeInstance that the QuantityInstance has in the corresponding State. A QuantityInstance is not guaranteed to have a ValueInstance or DerivativeInstance in every state (or even in every state in which the QuantityInstance itself exists), but the ValueInstance or DerivativeInstance that it has in a State is guaranteed to be one of the possible values or derivatives that the getPossibleValues and getPossibleDerivatives methods return.

QuantityInstance does not have methods to set the ValueInstance or DerivativeInstance in a particular State. This is determined by the simulator and is not something that a modeler will edit.

7.6 Values

Values exist in three forms in the gkom.model package: ValueType at the type-level, Value at the occurrence-level and ValueInstance at the instance-level.

7.6.1 ValueType

Class ValueType represents values at the type-level. A ValueType has a name and a value type: either point or interval. Unlike all the other child-classes of the Type class, ValueTypes are not members of a Model, but members of a QuantitySpace². Class QuantitySpace also serves as the builder for the ValueTypes that it contains, as describe in section 7.4. Every ValueType has a reference to the QuantitySpace it belongs to in its quantitySpace attribute.

ValueType
name: String quantitySpace: QuantitySpace valueType: int

Diagram 7-14: Class ValueType.

The valueType attribute refers to the value being a *point* or an *interval*.

7.6.2 Value

Values at the occurrence-level are represented by objects of the Value class. A Value is a member of a ModelFragmentType or ScenarioType, but is created by the Quantity it belongs to when that Quantity is assigned a QuantitySpace.

Value
getName: String getValueType: int
type: ValueType quantitativeValue: String quantity: Quantity

Diagram 7-15: Class Value.

A Value does not have an editable name: it is named after the ValueType it is an occurrence of. Likewise, the valueType attribute of a Value is also taken from the Value's type. The valueType

² To have a sensible response to the getModel method defined in the Type class, ValueType returns its QuantityType's Model.

attribute must not be confused with a Value's type attribute: the valueType attribute refers to the value being a point or an interval, the type attribute refers to the ValueType that the Value is an occurrence of. The quantitativeValue attribute was added to the Quantity class for completeness: it exists in GARP but was never actually used.

7.6.3 ValueInstance

Class ValueInstance, which represents values at the type-level, resembles class Value very closely. The only difference is that a ValueInstance belongs to a QuantityInstance and is created by one, whereas Values belong to and are created by Quantities.

ValueInstance
getName: String getValueType: int
type: ValueType quantitativeValue: String quantity: QuantityInstance

Diagram 7-16: Class ValueInstance.

7.7 Derivatives

Apart from having a value, a quantity may have a derivative. A derivative captures the 'direction' that the value of a quantity is moving in: it could be stable, increasing or decreasing. For consistency, an effort is made in GKOM to have derivatives behave in the same way that values do. Like values, derivatives are represented at all three knowledge levels. Quantities and QuantityInstances create and manage a set of Derivative and DerivativeInstance objects in the exact same way that they create and manage their set of Value and ValueInstance objects. The primary difference between values and derivatives is that there is no quantity space for derivatives, as is explained in the next section.

7.7.1 DerivativeType

In GARP, a quantity space that needs to be defined in every model is the derivative quantity space. This quantity space is always defined as min-zero-plus for decreasing, stable and increasing. Theoretically, a different quantity space for derivatives could be defined, but in practice the min-zero-plus space is always used.

In GKOM, the min-zero-plus quantity space is fixed for derivatives. Because of this, GKOM takes a slightly different approach. Instead of creating the min-zero-plus quantity space by default for every model or expecting a modeler to always start by defining a derivative quantity space, the quantity space for derivatives is implied. Every Quantity creates a min, zero and plus derivative for itself and does not allow the modeler to change that set.

The goal to have derivatives behave similar to the way values do calls for a DerivativeType class to perform the same role that the ValueType class does for values. This in turn would call for a separate class to represent a quantity space for derivatives, since the QuantitySpace class holds ValueTypes, not DerivativeTypes. At the same time, however, the need for such a specialized quantity space class is eliminated by the fact that GKOM will only use the min-zero-plus quantity space for derivatives: there is no need to create a class of which only one object will ever be created. In fact, there will never be a need for more than three DerivativeTypes: one for 'min', one for 'zero' and one for 'plus'. These considerations have led to the design of the DerivativeType class as depicted in Diagram 7-17.

DerivativeType
name: String derivativeType: int MIN: DerivativeType ZERO: DerivativeType PLUS: DerivativeType

Diagram 7-17: Class DerivativeType.

The MIN, ZERO and PLUS attributes of DerivativeType are static members of the class. This means that they are considered attributes of the class itself, not attributes of its objects (in which case they would be unique for every object and would only come into existence once an object is constructed). The attributes are themselves DerivativeType objects, namely the DerivativeTypes min, zero and plus. This technique³ makes sure that the three DerivativeType objects that will be used in Models are always available, and at the same time eliminates the need for a class representing a derivative quantity space, because the DerivativeType class itself fulfils the role of holding a set of DerivativeTypes.

The name-attribute holds the name of the DerivativeType. The derivativeType attribute has the same function as the valueType attribute of a ValueType: it identifies whether a DerivativeType is a point value or an interval value. The name of the MIN DerivativeType is 'min' and its derivative type attribute is 'interval'. PLUS is also an interval and is named 'plus'. ZERO is a point called 'zero'.

7.7.2 Derivative

Derivatives at the occurrence-level, represented by class Derivative, belong to a Quantity and are part of constructs. They can also be used as the left or right hand side of dependencies (section 7.8). A Derivative closely resembles a Value.

Derivative
getName: String getDerivativeType: int
type: DerivativeType quantity: Quantity

Diagram 7-18: Class Derivative.

Like Values, Derivatives get their name and their derivative type (value type for Values; either point or interval) from their type. A Derivative's DerivativeType is one of the three predefined static members of the DerivativeType class: MIN, ZERO and PLUS. Quantities create their own Derivatives at creation time, one for each of the three DerivativeTypes in class DerivativeType. The Quantity creating a Derivative sets the type- and quantity-attributes, which can thereafter not be altered by the modeler.

7.7.3 DerivativeInstance

Class DerivativeInstance represents derivatives at the instance-level. A DerivativeInstance closely resembles a ValueInstance, the main differences being that the type-attribute of a DerivativeInstance refers to a DerivativeType, not a ValueType. DerivativeTypes also differ just slightly from Derivatives: the quantity-attribute of a DerivativeInstance refers to a QuantityInstance rather than to a Quantity.

³ Having a class create an object of itself and keeping a static reference to it is a technique derived from the Singleton pattern (Gamma et al, 1995).

DerivativeInstance
getName: String getDerivativeType: int
type: DerivativeType quantity: QuantityInstance

Diagram 7-19: Class DerivativeInstance.

Like Derivatives, DerivativeInstances get their name and derivative type from their type. The QuantityInstance that creates the DerivativeInstance sets its type- and quantity-attribute, which thereafter are never altered.

7.8 Dependencies

Dependencies are relations between quantities, values or derivatives. They have a left and a right hand side, which can be a quantity, a value, a derivative or a nested dependency, depending on the dependency type. There is a fixed set of dependency types that can be separated into three categories: inequalities, correspondences and causal dependencies.

Because the set of dependency types is fixed, GKOM does not have classes to represent dependencies at the type-level. Such classes would only be useful if the modeler would be allowed to create new types of dependencies, which is not the case. Instead, every dependency has a dependencyType attribute, which is an integer representing one of the 20 dependency types. The individual types will be described in sections 7.9, 7.10 and 7.11.

Because of the similarity in their structure, all dependency types could have easily been treated as a single primitive of the GKOM knowledge representation. In that case there would have been a need for two classes only: one to represent dependencies at the occurrence-level and one to represent them at the instance-level. However, there are two good reasons for differentiating three categories of dependencies. One is that the types of dependencies in the three categories are semantically different. As sections 7.9 and 7.11 explain, the category of inequalities is very different from the category of causal dependencies. The second reason is that the types in the three categories behave differently: an inequality can have quantities, values, derivatives and nested inequalities as left and right hand side, while a causal dependency must have quantities at both sides. Separating between the three categories allows GKOM to restrict the usage of dependencies to prevent incorrect use.

To take advantage of the similarity of the dependency types and at the same time highlight their differences, GKOM defines an extra class at the occurrence and instance-level to act as a parent to those representing dependencies. Figure 7-2 shows all classes in the GKOM class hierarchy.

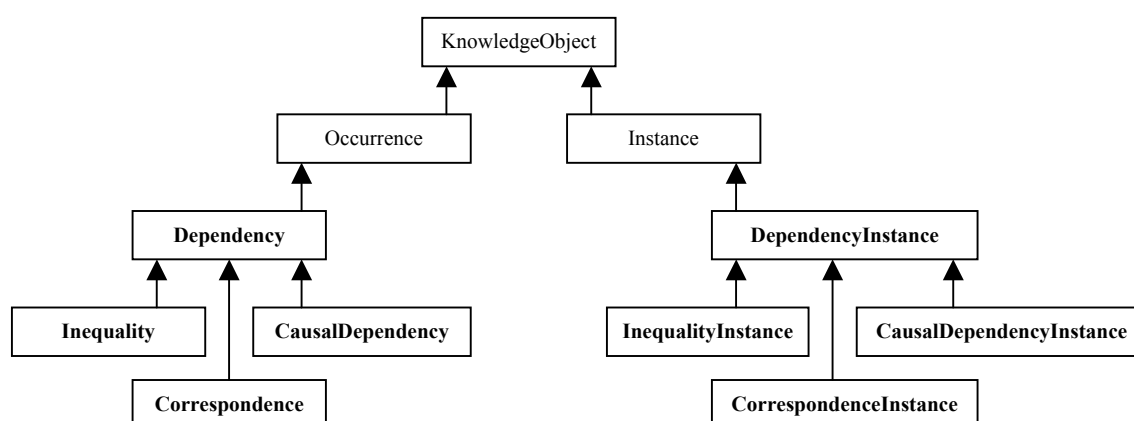


Figure 7-2: The dependency class hierarchy.

At the occurrence-level, classes Inequality, Correspondence and CausalDependency represent the three categories. All three inherit from class Dependency. At the instance-level, the three categories are represented by InequalityInstance, CorrespondenceInstance and CausalDependencyInstance,

which all inherit from `DependencyInstance`. Most of the functionality offered by the classes is part of the `Dependency` and `DependencyInstance` classes, thus taking advantage of the similarity between dependencies. Their child-classes contain functionality specific to the dependency categories.

The next two sections describe the `Dependency` and `DependencyInstance` classes in detail.

7.8.1 Dependency

Class `Dependency` is the parent-class of `Inequality`, `Correspondence` and `CausalDependency`. It contains the functionality that all dependencies at the occurrence-level have in common.

Dependency
contains: boolean
leftHandSide: Occurrence rightHandSide: Occurrence dependencyType: int

Diagram 7-20: Class `Dependency`.

Every `Dependency` has a left and right hand side. As explained in the above section, dependencies can have quantities, values, derivatives and nested inequalities on either side. In the case of `Dependency`, which is at the occurrence-level, this means that objects of the classes `Quantity`, `Value`, `Derivative` and `Inequality` can be at the left or right hand side of a `Dependency`. Since these four classes do not share a common parent-class more specific than class `Occurrence`, the left and right hand side attributes of `Dependency` are treated as objects of the `Occurrence` class. As sections 7.9.1, 7.10.1 and 7.11.1 explain, the child-classes of `Dependency` restrict what kinds of `Occurrences` can be set as the left or right hand side.

Apart from the left and right hand side, the child-classes of `Dependency` also inherit the `dependencyType` attribute. This attribute is an integer representing one of the 20 types of dependencies that exist across the three categories. The value of the `dependencyType` attribute of a `Dependency` is restricted by the dependency category: the three categories each encapsulate a subset of the 20 types.

Method `contains` is a utility method. It allows GKOM to check whether a particular `Quantity`, `Value` or `Derivative` is part of a particular dependency. GKOM will use this method when a modeler removes a quantity from a `ModelFragmentType` or `ScenarioType` to check whether removing the `Quantity` has an effect on other members of the `ModelFragmentType` or `ScenarioType`.

7.8.2 DependencyInstance

Class `DependencyInstance` is the parent-class of `InequalityInstance`, `CorrespondenceInstance` and `CausalDependencyInstance`. It contains the functionality that all dependencies at the instance-level have in common.

DependencyInstance
leftHandSide: Instance rightHandSide: Instance theDependencyType: int

Diagram 7-21: Class `DependencyInstance`.

The only difference between the `Dependency` class and the `DependencyInstance` class is that the left and right hand side of a `DependencyInstance` are instances rather than occurrences. The left and right hand side of a `DependencyInstance` can be objects of the classes `QuantityInstance`, `ValueInstance`, `DerivativeInstance` or `InequalityInstance`.

7.9 Inequalities

The Inequality classes (Inequality at the occurrence-level and InequalityInstance at the instance-level) represent the first of the three dependency categories introduced in section 7.8. The dependency types in this category are simple equality and inequality statements between the quantities, values and derivatives on the left and right hand side of a dependency. Inequalities represent such statements as ‘the value of quantity q is greater than zero’ or ‘the value of quantity q1 is equal to the value of quantity q2’. Additionally, an inequality can have a nested inequality at its left or right hand side, which allows qualitative calculus statements like ‘the value of quantity q1 is equal to the value of quantity q2 plus the value of quantity q3’.

Fundamentally, there are five (in)equality statements: ‘equal’, ‘greater’, ‘greater or equal’, ‘smaller’ and ‘smaller or equal’. The five inequality types with the corresponding names relate quantities and values, to represent statements like ‘the value of quantity q equals plus’, ‘the value of quantity q1 is greater than the value of quantity q2’ and ‘value max of quantity q1 is smaller than value max of quantity q2’. Another five inequality types relate quantities and derivatives: ‘derivative equal’, ‘derivative greater’, ‘derivative greater or equal’, ‘derivative smaller’ and ‘derivative smaller or equal’. Additionally, there are two nested inequality types: ‘plus’ and ‘min’. In total the category of inequalities contains twelve dependency types.

7.9.1 Inequality

The Inequality class represents inequalities at the occurrence-level. Inequality is a child-class of Dependency and inherits the leftHandSide, rightHandSide and dependencyType attributes from that class. The function of the Inequality class is to restrict the functionality of its parent-class to that which is specific for an Inequality. This is achieved by allowing only objects of the proper class as left or right hand side objects.

Inequality
setLeftHandSide getLeftHandSide: Occurrence setRightHandSide getRightHandSide: Occurrence

Diagram 7-22: Class Inequality.

As explained in section 7.8.1, the left and right hand side of a dependency are treated as occurrences. This follows from the fact that Java is a hard-typed language. In a hard-typed language, the programmer must be specific about the types of objects used. For example: a method that takes an Occurrence as an argument can never be passed an object of the String class; it must be given an object of the Occurrence class or of a class more specific than (i.e. down the hierarchy from) Occurrence. In the case of the Dependency class this presents a problem: both the leftHandSide and rightHandSide of a Dependency can be objects of four different classes: Quantity, Value, Derivative or Inequality. Dependency has to treat its left and right hand side attributes as objects of a class that all of these four classes inherit from. The most specific class that fits that description is class Occurrence, so Dependency treats its left and right hand side as objects of the Occurrence class. The problem lies in the fact that child-classes of Occurrence exist that are not valid as the left or right hand side of a Dependency, such as Entity or Relation. If a user would have direct access to these attributes, invalid Dependencies could be created.

Two things prevent a user from making objects of classes other than Quantity, Value, Derivative and Inequality the left or right hand side of an Inequality. One is that the leftHandSide and rightHandSide attributes of Dependency are protected, which means a user cannot directly access them. The other is that class Inequality gives access to these attributes in a restricted way: it does not allow the user to set the left or right hand side to an invalid object. It achieves this by defining four setLeftHandSide and four setRightHandSide methods: one that takes a Quantity as argument, one that

takes a Value, one that takes a Derivative and one that takes an Inequality⁴. The leftHandSide and rightHandSide attributes, inherited from Dependency, are still considered as objects of the Occurrence class, but because these attributes can only be edited through the methods described, they can simply not be set to an object of a class other than Quantity, Value, Derivative or Inequality.

The getLeftHandSide and getRightHandSide methods return a reference to the leftHandSide and rightHandSide attributes. Their return type is Occurrence, which can be cast back to the original type by the modeling application.

7.9.2 InequalityInstance

Class InequalityInstance represents Inequalities at the type-level. It is a child-class of DependencyInstance, from which it inherits the leftHandSide and rightHandSide attributes and the dependencyType attribute.

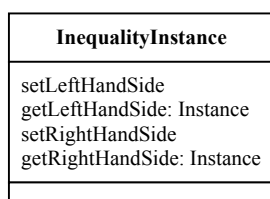


Diagram 7-23: Class InequalityInstance.

Like class Inequality, class InequalityInstance has four setLeftHandSide and four setRightHandSide methods. But these methods take objects of the classes QuantityInstance, ValueInstance, DerivativeInstance and InequalityInstance as arguments rather than their occurrence-level counterparts.

7.10 Correspondences

The second of the three dependency categories is Correspondence. Correspondences model how the value of one quantity can be inferred from the value of another. There are two types: *value correspondence* and *quantity space correspondence*. The former relates two individual values of two different quantities. The latter relates each of the values of two different quantities.

A Correspondence between two individual values of quantities represents that when one of the quantities is known to have a certain value, the other quantity must have the value on the other side of the correspondence. Two such correspondence types exist: ‘value correspondence’ and ‘directed value correspondence’. As the names imply, a ‘value correspondence’ holds in both directions (when either of the values is known, the other can be inferred) and a ‘directed value correspondence’ only holds in one direction only (when the left hand side value is known, the right hand side value can be inferred, but when the right hand side value is known the left hand side value cannot).

In GKOM, quantity space correspondences are represented by a correspondence that binds two quantities. This means that each of the values in the two quantity’s quantity spaces correspond. There are two such Correspondence types: ‘quantity space correspondence’ and ‘directed quantity space correspondence’. Like the value correspondences above, the former holds in both directions while the latter holds from left to right only. For two quantities to be in a quantity space correspondence together, both quantities must have the same quantity space.

Two classes exist in GKOM that represent Correspondences: class Correspondence at the occurrence-level and class CorrespondenceInstance at the instance-level.

7.10.1 Correspondence

Class Correspondence represents the dependency-category of correspondences at the occurrence-level. It is a direct child-class of Dependency and inherits the leftHandSide, rightHandSide and

⁴ In the Object Oriented Paradigm this technique is referred to as ‘method overloading’: defining several methods in one class that have the same name but take arguments of a different kind.

dependencyType attributes from that class. Its purpose is to offer functionality specific to Correspondences.

Correspondence
setLeftHandSide getLeftHandSide: Occurrence setRightHandSide getRightHandSide: Occurrence

Diagram 7-24: Class Correspondence.

Like class Inequality (see section 7.9.1), class Correspondence gives access to the leftHandSide and rightHandSide attributes of Dependency in a way that makes it impossible for the modeler to create Correspondences with invalid Occurrences as their left or right hand side. Only two types of occurrences can be the left and right hand side of Correspondences: Quantity and Value. Consequentially, there are two setLeftHandSide and two setRightHandSide methods: one that takes a Quantity as argument and one that takes a Value. The getLeftHandSide and getRightHandSide methods return the left and right hand side as an occurrence, which must be cast back to the original type by the modeling application.

7.10.2 CorrespondenceInstance

Class CorrespondenceInstance represents correspondences at the instance-level. As a child-class of DependencyInstance, it inherits the leftHandSide, rightHandSide and dependencyType attributes from that class.

CorrespondenceInstance
setLeftHandSide getLeftHandSide: Instance setRightHandSide getRightHandSide: Instance

Diagram 7-25: Class CorrespondenceInstance.

CorrespondenceInstance has two setLeftHandSide methods and two setRightHandSide methods; one that takes an object of the QuantityInstance class as an argument and one that takes a ValueInstance as an argument. The getLeftHandSide and getRightHandSide methods return the left and right hand side attributes more generally as an Instance.

7.11 CausalDependencies

The last of the three categories of Dependencies is CausalDependency. CausalDependencies model how quantities influence each other. There are two different causalities: proportionalities and influences. Both can be positive or negative, creating four different causal dependency types. Proportionalities model that a change in one quantity causes a change in another quantity in the same direction ('positive proportionality') or in the opposite direction ('negative proportionality'). Influences model how the value of one quantity influences the derivative of another, in either the same direction ('positive influence'; if the value is positive, the other quantity increases, if the value is negative it decreases) or the opposite direction ('negative influence'; if the value is positive, the other quantity decreases, if the value is negative it increases).

Because of the causal nature of this type of dependencies, the terms left hand side and right hand side are renamed to influencer and influenced in correspondences.

Two classes are used to represent causal dependencies: class CausalDependency and class CausalDependencyInstance.

7.11.1 CausalDependency

At the occurrence-level, class `CausalDependency` represents the dependencies that belong to the category of causal dependencies. `CausalDependency` is a child-class of `Dependency` and inherits its `leftHandSide`, `rightHandSide` and `dependencyType` attributes.

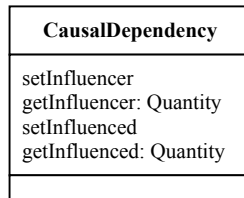


Diagram 7-26: Class `CausalDependency`.

A `CausalDependency` refers to the `leftHandSide` and `rightHandSide` attributes inherited from its parent-class as its influencer and its influenced. The `setInfluencer` and `getInfluencer` methods operate on the `leftHandSide` attribute, while the `setInfluenced` and `getInfluenced` methods operate on the `rightHandSide` attribute.

`CausalDependencies` relate objects of the `Quantity` class only. Where `Inequality` needs four methods to assign the left and right hand side (for `Quantities`, `Values`, `Derivatives` and `Inequalities`) and `Correspondence` needs three (for `Quantities`, `Values` and `Derivatives`), `CausalDependency` needs only a single method to set the influencer and a single method to set the influenced. Because there is no ambiguity about what types of objects are related by `CausalDependency`, the `getInfluencer` and `getInfluenced` methods return objects of the `Quantity` class; there is no need to return them as `Occurrence` objects, as the `Inequality` and `Correspondence` classes do.

The fact that both `Influencer` and `Influenced` are objects of the `Quantity` class also means that `CausalDependency` does not need to offer any modeling support. The `Inequality` and `Correspondence` classes offer support when a conflict arises between the dependency type and the type of left and right hand side objects used. Because only one type of object is used as influencer and influenced of a `CausalDependency`, a modeler cannot make such mistakes when working with `CausalDependencies`.

7.11.2 CausalDependencyInstance

The `CausalDependency` class represents the category of causal dependencies at the instance-level. `CausalDependency` is a child-class of the `DependencyInstance` class.

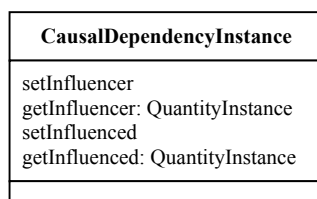


Diagram 7-27: Class `CausalDependencyInstance`.

Like `CausalDependency`, `CausalDependencyInstance` refers to its left hand side as ‘influencer’ and to its right hand side as ‘influenced’. Both are objects of the `QuantityInstance` class.

The model ingredients described in this chapter are the building blocks with which scenarios and model fragments are created. The next chapter describes these.

8 Model fragments and scenarios

Model fragments and scenarios are aggregates of the building blocks described in the previous chapter. Model fragments capture behavioral aspects that emerge when several individual building blocks are grouped together. Scenarios describe a real-world system to serve as initial state for a simulation process. In GKOM, model fragments and scenarios exist at the both the type-level, where their members are occurrences, and the instance-level, where their members are instances. At the type-level model fragments and scenarios play a role in modeling support by acting as builders for their members and by actively ensuring their own validity.

In this chapter, the first section explains how model fragments and scenarios act as builders for their members. Subsequently, the classes that represent model fragments and scenarios in the GKOM module are described.

Model fragments and scenarios as builders

Model fragments and scenarios at the type-level are the builders for their own members, in the same way that the Model class builds its members. There are two reasons for this, one related to offering support and one to maintaining consistency in a Model. The first reason is that a lot of the support offered by GKOM revolves around the fact that the type-level is used to define possible behavior at the occurrence-level. The type-level contains ontological information: it captures what types of things can exist in a model and how these things can be interrelated. GKOM enforces consistency between the type and occurrence-level by taking the type-level as authoritative and notifying the modeler whenever the modeler tries to carry out an operation on an occurrence that its type will not allow. From this it follows that inconsistencies between the type and occurrence-levels can never be introduced in a Model; GKOM will not allow it. But that remains true only as long as the type attribute of occurrences is not changed. If, for example, a modeler would be allowed to change the QuantityType of a Quantity in a ModelFragmentType or ScenarioType *after* that Quantity has been added to an Entity, the Quantity could become invalid because its new type does not allow it to belong to the Entity that it was created for. Furthermore, the Quantity's QuantitySpace may no longer be applicable. One way to deal with this situation would be to simply remove the QuantitySpace and its Values from the Quantity, but that would in turn invalidate any Dependencies referring to these Values. Hence the Values and Dependencies must also be removed. In short, allowing a modeler to change the type of an occurrence can have a cascading effect that is very hard to foresee.

These considerations have led to the decision that the type attribute of occurrences should not be user editable. The preferred way to do this in Java is making the setType method 'protected', which means that it cannot be called from outside the gkom.model package. This calls for helper methods inside the gkom.model package that create occurrences for the user, since creating an occurrence requires calling its setType method. These methods are implemented by all of the constructs, making the constructs the *builders* of the occurrences they contain. The builder pattern is described in Gamma (Gamma et al, 1995).

The second reason for making constructs the builders of their members is that adding members to and removing members from a construct usually requires multiple steps. To add a Property to a construct, for example, the Property must be added to the construct and to an Entity in the construct. Removing the Property is also a two-step process: the Property must be removed from the construct and from its Entity. In either case, omitting one of the two steps makes the construct as a whole inconsistent: it would lead to either a construct containing an Entity that has a Property which is not part of the construct or to a construct containing a Property that does not belong to an Entity.

As builders, constructs can prevent such errors from occurring by implementing methods that perform the entire process of adding and removing occurrences. The create methods of the constructs handle creating occurrences, setting their type, adding them to the constructs and adding them to other occurrences in the construct. The remove methods remove the occurrences from other occurrences and remove the occurrences from the construct it was part of.

8.1 Scenarios

Scenarios describe the initial state of a simulated system. They are the starting point of a simulation and change throughout it. In GKOM, class `ScenarioType` describes an initial state fed to the simulator and the `ScenarioInstance` class represents the state of a `ScenarioType` at a particular point in the simulation.

8.1.1 ScenarioType

Class `ScenarioType` is a child of the `Type` class and represent a scenario at the type-level. A `Model` can hold any number of `ScenarioTypes`, but each must have a unique name. `ScenarioTypes` are also used to uniquely identify `Simulations` in a `Model`: there can be one `Simulation` for every `ScenarioType` in a `Model`.

ScenarioType
createMember: Occurrence getMember: Occurrence getMembers: Occurrence[] removeMember: boolean
entities: Entity[] properties: Property[] relations: Relation[] quantities: Quantity[] values: Value[] derivatives: Derivative[] inequalities: Inequality[] name: String

Diagram 8-1: Class ScenarioType.

The `createMember`, `getMember`, `getMembers` and `removeMember` methods shown in Diagram 8-1 are not the actual methods of the class, but represent methods of that signature for each of the types of objects that a `ScenarioType` can contain. A `ScenarioType` can contain objects of the `Entity`, `Property`, `Relation`, `Quantity`, `Value`, `Derivative` and `Inequality` classes, so there are create, get and remove methods for each of those.

With its create and remove methods, `ScenarioType` implements its role as builder for its members. The create methods handle creating a new member, adding it to the construct and possible adding it to other occurrences in the construct. To do this properly, the create methods require all the relevant information as arguments:

- The `createEntity` method takes an `EntityType` and the name for the new `Entity` as arguments. With these, the `ScenarioType` creates a new `Entity`, sets its `EntityType` and name and adds the `Entity` to itself.
- The `createProperty` method takes a `PropertyType`, an `Entity` and a value name as arguments. The `Entity` must already be part of the `ScenarioType`. The method creates a new `Property`, sets its type and value, adds it to the `Entity` given and finally adds the `Property` to the `ScenarioType`.
- The `createRelation` method takes a `RelationType` and a left and right hand side `Entity` as arguments. The `Entities` must already be part of the construct. The method creates a new `Relation`, sets its `RelationType`, sets the left and right hand side entities and then adds the `Relation` to the `ScenarioType`.
- The `createQuantity` method receives three arguments: a `QuantityType`, a name for the `Quantity` and an `Entity` to which the `Quantity` will belong. The `Entity` must already be part of the `ScenarioType`. The method creates a new `Quantity`, sets its type and name and adds it to the `Entity`. Then the new `Quantity` is added to the `ScenarioType`.
- The `createValue` method takes as arguments a `Quantity` and the name of one of the `Values` of that `Quantity`. The `Quantity` must already be part of the `ScenarioType`. The method does not actually create the `Value` object, since a `Quantity` creates its own `Values` when given a

QuantitySpace. Instead, the method gets the Value by the specified name from the Quantity, sets that value as the Quantity's current value and adds the Value object to the ScenarioType.

- The createDerivative method works the same way that the createValue method does. It takes a Quantity object and a derivative value name as arguments and uses the derivative value name to retrieve the corresponding Derivative object from the Quantity. It subsequently sets that Derivative as the Quantity's current Derivative and adds the Derivative to the ScenarioType.
- The createInequality method takes only a single argument: an inequality type. It creates a new Inequality of the given type and adds it to the ScenarioType. Setting the Inequality's left and right hand side is left up to the modeler.

For each of the create methods described here there is a corresponding remove method. The remove methods all take a single argument: the occurrence to be removed. A remove method removes the occurrence given from the construct and from any other occurrences in the construct that it might be related to.

The only way to edit the sets of members of a ScenarioType is through its create and remove methods. The attributes entities, properties, relations, quantities, values, derivatives and inequalities shown in Diagram 8-1 are private to a ScenarioType.

8.1.2 ScenarioInstance

Every State in a Simulation has a ScenarioInstance object, which represents the current state of the ScenarioType that is the basis of the Simulation. A ScenarioInstance knows the ScenarioType it represents the current state of: it is stored in its type attribute.

ScenarioInstance is the only instance that is recreated for every state individually and is always a part of only one state. All other instances are created once per Simulation and can be part of multiple states, as explained in section 6.2.6. ScenarioInstance is different because it is its function to represent a changing 'system' in each state: every ScenarioInstance is different.

ScenarioInstance
addMember getMembers: Instance[] removeMember: boolean
entities: EntityInstance [] properties: PropertyInstance [] relations: RelationInstance [] quantities: QuantityInstance [] values: ValueInstance [] derivatives: DerivativeInstance [] inequalities: InequalityInstance [] state: State

Diagram 8-2: Class ScenarioInstance

Class ScenarioInstance is very similar to class ScenarioType. There are three differences:

- A ScenarioInstance has instances as members.
- A ScenarioInstance does not act as a builder for its members. Instances are created by a Simulation object, as explained in section 6.2.6, and added to ScenarioInstances by the parser.
- A ScenarioInstance has a state attribute. The state attribute is simply a reference to the State that the ScenarioInstance is part of.

There are add, get and remove methods for each of the different kinds of instances that a ScenarioInstance can contain. There are no methods to retrieve instances individually; the get methods (getEntities, getProperties, getRelations, getQuantities, getValues, getDerivatives and getInequalities) retrieve all instances of a particular kind. Individual instances in a Simulation can be retrieved from the Simulation itself.

8.2 Model Fragments

Model fragments exist at all three knowledge levels in GKOM: `ModelFragmentType` at the type-level, `ModelFragment` at the occurrence-level and `ModelFragmentInstance` at the instance-level. The occurrence-level representation is used to represent model fragments nested in `ModelFragmentTypes`. Both `ModelFragmentTypes` and `ModelFragmentInstances` have two sets of members: `Conditions` and `Givens`.

Parent model fragments and embedded model fragments

In GARP, model fragments are embedded in a hierarchy. The model fragment hierarchy, unlike the entity hierarchy, supports multiple inheritance: a model fragments can have several parents. Additionally, model fragments can be embedded in other model fragments, which is a different principle entirely. These two features complicate the representation of model fragments in GKOM. Before going into details about the structure and functionality of the classes representing model fragments, this section will describe these two features in general terms.

A model fragment consists of a set of conditions and a set of givens. Both sets contain interrelated model ingredients. One of the steps performed by the simulator is to match all the givens blocks of all the model fragment types in the library against the current system description. If it finds a match, it adds the model ingredients in the givens block of the matching model fragment to the current system description.

When one model fragment is the child of another model fragment, the simulator will match the conditions blocks of both model fragments against the current system description before it can determine that the child model fragment should be activated. The child model fragment inherits all entities and quantities from its parent and can add a set of conditions of its own, or further refine the constraints upon the entities it inherits. The latter would be the case if the child model fragment would specify that an entity it inherits from its parent must be of a more specific type (e.g. a ‘gas’ instead of a ‘substance’) in order for the child model fragment to become active.

Apart from parents, a model fragment can have nested model fragments, which are part of its conditions block. There is a subtle but important difference between a parent model fragment and a nested model fragment. As with its parents, a model fragment will only become active in a simulation if its nested model fragments are active. But a model fragment does not inherit the entities from its nested model fragments: the nested model fragments serve to specify an additional set of conditions over entities defined in the model fragment in which they are nested. A nested model fragment could be used, for example, to specify that the container in model fragment ‘contained liquid’ must be ‘like’ the container used in model fragment ‘open container’: the conditions specified in ‘open container’ must be met for the container in ‘contained liquid’. There may be several nested model fragments specifying different conditions over one entity in the containing model fragment and there may be several nested model fragments of the same type specifying the same conditions over different entities in the containing model fragment.

These two features have proven to be difficult to represent in the GKOM model. The problem is that an entity in GKOM (and in fact all other model ingredients) can be viewed from only one perspective. Every entity has a single type: it cannot have one type in one model fragment and another in another model fragment. The same holds for its name: an entity cannot have two names in two different model fragments. Additionally it also has a single set of properties, relations and quantities: from an entity’s point of view there is no way to discriminate between properties, relations and quantities it has in one model fragment and properties, relations and quantities it has in another. An entity can in fact not belong to two model fragments at the same time.

GKOM solves this problem by creating copies of entities: an entity in a model fragment that is ‘inherited’ from a parent model fragment is actually a different object. This second object can be used to specify further constraints on: it can have a different type than the original; it can have a different name and it can have a set of properties, relations and quantities of its own. The original object is not changed. The model fragment knows that the original and the copy are related and can make sure that the modeler does not specify conflicting constraints (e.g. giving an entity in the child model fragment a more general type than the entity in the parent) and does not perform any actions that would lead to

inconsistencies between model fragments (e.g. removing an entity from a model fragment for which a copy exists in a nested model fragment).

However, when a model fragment is nested inside another model fragment, GKOM cannot simply start making copies of all the entities in the nested model fragment. This is because two nested model fragments can be used to specify additional constraints over a single entity in the containing model fragment. In that situation, if all entities in both nested model fragments would simply be copied, the containing model fragment would end up with two objects to represent the same entity. Also, a containing model fragment may not reuse all entities in its nested model fragments, so simply copying all entities from them could result in creating unnecessary objects. In both cases, additional user interaction would be needed to either ‘merge’ two entities into one or remove unnecessary entities.

GKOM solves the problems and difficulties described above by taking a different view on nesting model fragments. Instead of thinking of a nested model fragment as the result of adding a model fragment to the conditions block of another model fragment, it thinks of it as the result of specifying that an entity in one model fragment has a ‘peer’ in another model fragment. If a modeler would want to say that the container in ‘contained liquid’ must adhere to the same conditions as the container in ‘open container’, the modeler would create a container in ‘contained liquid’ and specify that the container in ‘open container’ is its peer. The result of this is that ‘open container’ is nested in ‘contained liquid’ and that the container in ‘contained liquid’ is considered a copy of the container in ‘open container’. One entity in a model fragment can be given several different peers, which will result in the model fragment having several different nested model fragments. And several entities in a model fragment can be given the same peer, which will result in the model fragment having several nested model fragments of the same type.

In the same way that nested model fragments can be considered the result of specifying that an entity in the containing model fragment is a peer of an Entity in the nested model fragment, parent model fragments can be considered the result of specifying that an entity in a parent model fragment is a parent of an entity in a child model fragment. If a modeler makes such a specification, the model fragment in which the child entity resides becomes a child of the model fragment in which the parent model fragment resides and the child entity is considered a copy of the parent entity. But where a single entity can have multiple peers, it can only have one parent. Multiple inheritance for model fragments is still possible, however: since every entity in a model fragment can have a parent, the model fragment as a whole can have multiple parents.

8.2.1 ModelFragmentType

A ModelFragmentType, which represents a model fragment at the type-level, has a name, a Conditions and Givens block and optionally one or more parent and child ModelFragmentTypes. ModelFragmentTypes are part of a Model, in which context they have a unique name.

ModelFragmentType
getConditions: ModelFragmentType.Conditions getGivens: ModelFragmentType.Givens createMember: Occurrence. containsMember: boolean getMember: Occurrence getMembers: Occurrence[] removeMember: boolean getParents: ModelFragmentType[] getChildren: ModelFragmentType[] hasChild: boolean specifyParentEntity specifyPeerEntity unspecifyParentEntity unspecifyPeerEntity
mfType: int

Diagram 8-3: Class ModelFragmentType.

The `getConditions` and `getGivens` methods of a `ModelFragmentType` return its Conditions and Givens block, respectively¹. The two blocks resemble each other closely: each contains a set of Entities, Properties, Relations, Quantities, Values and Derivatives. In addition the Conditions block contains a set of Inequalities and a set of nested model fragments (Nested model fragments are represented by `ModelFragment` objects) and the Givens block contains a set of Dependencies.

ModelFragmentType.Conditions	ModelFragmentType.Givens
<code>addMember</code> <code>getMember: Occurrence</code> <code>getMembers: Occurrence[]</code> <code>removeMember: boolean</code>	<code>addMember</code> <code>getMember: Occurrence</code> <code>getMembers: Occurrence[]</code> <code>removeMember: boolean</code>
<code>entities: Entity[]</code> <code>properties: Property[]</code> <code>relations: Relation[]</code> <code>quantities: Quantity[]</code> <code>values: Value[]</code> <code>derivatives: Derivative[]</code> <code>inequalities: Inequality[]</code> <code>modelFragments: ModelFragment[]</code>	<code>entities: Entity[]</code> <code>properties: Property[]</code> <code>relations: Relation[]</code> <code>quantities: Quantity[]</code> <code>values: Value[]</code> <code>derivatives: Derivative[]</code> <code>dependencies: Dependency[]</code>

Diagram 8-4: Inner classes `ModelFragmentType.Conditions` and `ModelFragmentType.Givens`.

The `add`, `get` and `removeMember` methods in Diagram 8-4 represent methods in the actual classes that add, get and remove the different types of members. The `add` and `remove` methods are not available outside the `gkom.model` package, but are instead called by `ModelFragmentType` from its `create` and `remove` methods.

A `ModelFragmentType` acts as a builder for its own members. The class has nine create methods: `createEntity`, `createProperty`, `createRelation`, `createQuantity`, `createValue`, `createDerivative`, `createInequality`, `createCorrespondence` and `createCausalDependency`. The first seven of those take the same arguments as their counterparts in the `ScenarioType` class, with one additional argument: the context. Based on the context, the member is created in either the Conditions or the Givens block. Only the `createCorrespondence` and `createCausalDependency` methods do not take a context argument; they can only exist in the Givens block. Like the `createInequality` method in class `ScenarioType`, the methods take a dependency type as argument. The `get` and `removeMember` methods of `ModelFragmentType` have the same function and method signature as the corresponding methods in `ScenarioType`. They will get and remove members from both the Conditions and the Givens.

Class `ModelFragmentType` does not have methods to add, create or remove nested model fragments or parent model fragments. As explained in section 8.2, GKOM takes a different approach to adding parent and nested model fragments. Class `ModelFragmentType` implements this approach in its `specifyParentEntity` and `unspecifyParentEntity` methods and its `specifyPeerEntity` and `unspecifyPeerEntity` methods. All four of these methods take two Entities as arguments: a local Entity and a remote Entity. The local Entity must already be present in the Conditions block of the `ModelFragmentType` on which the method is called; the remote Entity must be a member of another `ModelFragmentType`.

By calling the `specifyParentEntity` method, the modeler specifies that the remote entity is a parent of the local entity. The result of this is that the `ModelFragmentType` in which the remote Entity resides becomes a parent of the `ModelFragmentType` that was called, and that the local Entity is considered a copy of the remote Entity. The `unspecifyParentEntity` does the reverse: the local Entity is no longer considered a copy of the remote Entity and the `ModelFragmentType` called is no longer a child of the `ModelFragmentType` in which the remote Entity resides.

The `specifyPeerEntity` and `unspecifyPeerEntity` methods work the same way, but add and remove nested model fragments instead of parent model fragments. The `specify` method can be called multiple times with the same local Entity but with remote Entities from different `ModelFragmentTypes`. The result of this is one local Entity that appears in two nested model fragments.

¹ The Conditions and Givens blocks are inner classes of the `ModelFragmentType` class, so their full names are `ModelFragmentType.Conditions` and `ModelFragmentType.Givens`.

Parent and nested model fragments are thus not added and removed directly, but appear and disappear as a result of calling the methods discussed above. They can, however, be queried. The `getParents` and `getChildren` methods of class `ModelFragmentType` will return all parent `ModelFragmentTypes` and children `ModelFragmentTypes`. The `hasChild` method is for convenience and tests whether a given `ModelFragmentType` is one of the children of the `ModelFragmentType` on which the method is called.

Nested model fragments can be queried through the `Conditions` block they are part of. Method `getModelFragment` returns a `ModelFragment` object by `ModelFragmentType` and method `getModelFragments` return all nested `ModelFragments` of a `ModelFragmentType`. The `ModelFragment` class is the subject of section 8.2.2.

ModelFragmentType's role in support

As a builder, `ModelFragmentType` offers the same type of support as the `ScenarioType` class does. In addition, it offers support related to the model fragment hierarchy and to nested model fragments.

Method `specifyParentEntity` of a `ModelFragmentType` throws an `InvalidOccurrenceException` in the following situations:

- If the local `Entity` supplied is not part of the `ModelFragmentType`'s `Conditions` block. This ensures that the local `Entity` is indeed local and that it is part of the `Conditions` block. Entities in the `Givens` block cannot have a parent.
- If the parent `Entity` is not part of a `ModelFragmentType`. An `Entity` that is a member of some other construct cannot be the parent of an `Entity` in a `ModelFragmentType`.
- If the type of the local `Entity` is not the same or more specific as the parent `Entity`'s type. A child `Entity` can be more restrictive, but not less restrictive.
- If the parent `Entity` is already a parent of another `Entity` in this `ModelFragmentType`. No `Entity` can be the parent of multiple `Entities` in one `ModelFragmentType`.

The `specifyPeerEntity` method of a `ModelFragmentType` throws an `InvalidOccurrenceException` in the following situations:

- If the local `Entity` supplied is not part of the `ModelFragmentType`'s `Conditions` block. This ensures that the local `Entity` is indeed local and that it is part of the `Conditions` block. Entities in the `Givens` block cannot have a peer.
- If the peer `Entity` is not part of a `ModelFragmentType`.
- If the type of the local `Entity` is not the same or more specific as the peer `Entity`'s type. A peer `Entity` can be more restrictive, but not less restrictive.

The `removeEntity` method of a `ModelFragmentType` performs the same checks as the `removeEntity` method in `ScenarioType` and performs one additional check. When the modeler tries to remove an `Entity`, the `ModelFragmentType` checks whether that `Entity` has a copy in other `ModelFragmentTypes` as either a parent or a peer. If so, it throws an `InconsistentModelException`. It does this regardless of the `force` option that can be used to instruct a construct to solve any inconsistencies that may result from an action. The `force` option is described in section 8.1.1. Although it would theoretically be possible to solve the inconsistencies resulting from the removal of an `Entity` that is a parent or a peer, the scope of such an action would be greater than just the `ModelFragmentType` called. GKOM will not do this; it will simply make the modeler aware of the problem by throwing an exception.

8.2.2 ModelFragment

The members of constructs at the type-level are all subtypes of `Occurrence`. The reason for having an occurrence-level in GKOM is in fact to underscore the fundamental difference between elements that define ontology-like information and specific uses of elements from the ontology inside constructs. The primary reason for creating an occurrence-level representation of model fragments is that model fragments can themselves be 'members' of other model fragments. For consistency, all members of `ModelFragmentTypes` should be occurrences.

`ModelFragments` are occurrences of `ModelFragmentTypes`. When an `Entity` in a `ModelFragmentType` is given a peer (through the `specifyPeerEntity` method described above) the

ModelFragmentType called creates a ModelFragment object whose type is the ModelFragmentType that the peer resides in². Subsequently it moves the Entity that has been given a peer into the newly created ModelFragment object, along with a reference to the peer. Then it adds the ModelFragment object to its Conditions block.

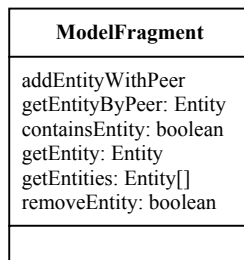


Diagram 8-5: Class ModelFragment.

To add the local Entity and the peer Entity to the ModelFragmentType, method addEntityWithPeer is used. Receiving both Entities allows the ModelFragment to establish a link between the two. With this link, Entities can be retrieved from a ModelFragment in two ways: either directly by name (method getEntity) or indirectly by peer (method getEntityByPeer). The latter method is used by ModelFragmentTypes to find copies made of their Entities: when the removeEntity method of a ModelFragmentType is called, it scans all its occurrences (ModelFragments that represent itself, nested in other ModelFragmentTypes) to see if any of their getEntityByPeer methods return any Entities. If it finds any Entities it will conclude that the Entity cannot be removed and throws an InconsistentModelException, to which it appends all Entities it has found. The Modeler can inspect the list to view all Entities that cause a conflict.

Method getEntities returns all Entities that a ModelFragment holds and method containsEntity is a convenience method for checking whether a given Entity is part of that set. The removeEntity method removes an Entity from a ModelFragment. It cannot be called outside the gkom.model package, but is called by the ModelFragmentType it is part of. This is also true for the addEntityWithPeer method.

ModelFragment does not have any attributes other than its type, which it inherits from class Occurrence.

8.2.3 ModelFragmentInstance

Class ModelFragmentInstance represents instances at the type-level. Its structure differs little from the ModelFragmentType class, except that all its members are themselves instances. It is not the builder of its own members, since at the instance-level the Simulation class creates all elements. As all instances can, a ModelFragmentInstance can be part of several states in a single Simulation.

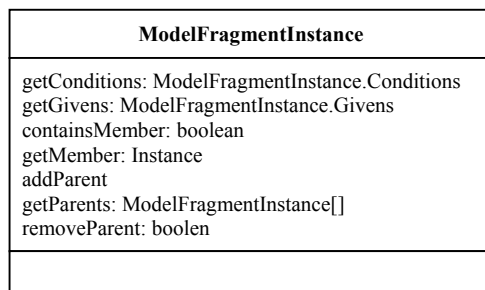


Diagram 8-6: Class ModelFragmentInstance.

² The ModelFragmentType will in fact first check whether or not it has already created the ModelFragment in a previous call to specifyPeerEntity.

Because a `ModelFragmentInstance` is not a builder, adding and removing members is somewhat simpler than in the `ModelFragmentType` class. Adding and removing members of the Conditions and Givens blocks can be done by calling the corresponding add and remove methods directly. This also goes for nested model fragments. Parent model fragments can be added directly to the `ModelFragmentInstance` itself. At the instance-level, this does not cause any trouble, because only a single object is created for all elements in a Simulation. If an `EntityInstance` in one `ModelFragmentInstance` has a parent in another, the two are actually one object. This is exactly what model fragments aim to represent, but what cannot be represented at the type-level, since the parent and the child can have different types, different names and different sets of properties, relations and quantities. At the instance-level this problem does not occur: both parent and child are one `EntityInstance` object in the simulation (with one `EntityType`, one name and one set of `PropertyInstances`, `RelationInstances` and `QuantityInstances`) that has caused two `ModelFragmentInstances` to become active.

Without the need to create copies, there is also no need to check for duplicate or unnecessary copies. And since the simulator can be trusted to do its work correctly, there is also no reason to disallow simply adding parent and nested model fragments directly to a `ModelFragmentInstance` object.

ModelFragmentInstance.Conditions	ModelFragmentInstance.Givens
addMember getMember: Instance getMembers: Instance[] removeMember: boolean	addMember getMember: Instance getMembers: Instance[] removeMember: boolean
entities: EntityInstance[] properties: PropertyInstance[] relations: RelationInstance[] quantities: QuantityInstance[] values: Value Instance[] derivatives: DerivativeInstance[] inequalities: InequalityInstance[] modelFragments: ModelFragmentInstance[]	entities: EntityInstance[] properties: PropertyInstance[] relations: RelationInstance[] quantities: QuantityInstance[] values: Value Instance[] derivatives: DerivativeInstance[] dependencies: DependencyInstance[]

Diagram 8-7: Classes `ModelFragmentInstance.Conditions` and `ModelFragmentInstance.Givens`.

The `ModelFragmentInstance.Conditions` and `ModelFragmentInstance.Givens` classes have the same methods and structure as the `ModelFragmentType.Conditions` and `ModelFragmentType.Givens` classes. The only two differences are that at the `ModelFragmentInstance.Conditions` and `ModelFragmentInstance.Givens` classes contain instances as members and that the add- and removeMember methods of these classes are public rather than only available in the `gkom.model` package.

8.3 Conclusion

This chapter and chapter 6 and 7 have given a detailed description of the implementation of the GKOM knowledge representation. Although these chapters are written from a developers' perspective, application builders will require even more detailed information. The technical documentation of the GKOM API will be made available at <http://www.swi.psy.uva.nl/projects/GARP/index.html>.

9 Conclusions and discussion

The aim of the GKOM module is to make it easier for application developers to build knowledge articulation tools for use in educational software. Developers can build their applications on top of the GKOM module and benefit from GKOM's knowledge representation, modeling support and networking capabilities. The GKOM module is currently being used in three separate applications:

- A modeling environment called MoBum (Bessa Machado, 2000). MoBum was developed before GKOM as a graphical modeling environment for creating GARP models. It has been successfully altered to use GKOM as its knowledge representation language.
- Model specification components developed by Groen (2003). Groen's components allow modelers to specify knowledge in an informal manner initially, to subsequently work down to the formal level of the GKOM knowledge representation.
- A simulation browsing application called GARPApplet. Like MoBum, the GARPApplet is an existing application that was later modified to use GKOM.

These three applications utilize different GKOM features. MoBum uses the knowledge representation and modeling support offered by GKOM, but does not communicate with the simulation service. Contrarily, the GARPApplet does not offer any modeling functionality, but focuses entirely on simulation. Groen's model specification tools do both.

The conclusions and recommendations in this chapter are based on the experience with implementing GKOM in the three applications mentioned above and on the considerations that we have developed in the process of creating the GKOM module.

9.1 Conclusions

In chapter 2, two objectives for building the GKOM module were identified. The first was to offer application builders a library of functions that facilitate the creation of educational modeling applications, in order to allow the application builders to focus on user interface and user interaction. The second was to create a component in a framework for collaborative learning and modeling. This section discusses each in turn.

Regarding the usability of the GKOM module as a basis for a modeling application, the experiences are positive. GKOM is currently used in several applications with good results, and the reactions of the application builders are positive. In the experiments that the application builders undertook with their applications, no serious problems related to GKOM have surfaced. Using GKOM as basis for their applications has given developers time to focus on user interface and user interaction rather than on system-oriented features such as the exact representation of knowledge.

As a knowledge representation, GKOM is more formal than GARP is, for two reasons. First, the entity hierarchy has been extended to include generic properties, relations and quantities. This allows modelers to express facts such as that certain types of entities have certain types of properties and that certain types of quantities go with certain types of entities. Second, the GKOM knowledge representation requires strict consistency between what is expressed at a generic level, using type-level elements, and what is expressed inside model fragments and scenarios. GKOM offers language constructs that allow modelers to express their beliefs about a domain, but at the same time confronts modelers with the consequences of their choices. We believe that this feature will prove valuable in educational settings.

The modeling support facilities that GKOM offers are being put to good use in the applications built so far. This may in part be due to the practice, described in section 5.5, of using *exceptions* to convey support information, which forces application developers to handle the modeling support feedback one way or the other. The simplest way for a developer to handle an exception is to display the exception message to the user. But the GKOM exceptions contain more information than a message: they also hold references to the elements that cause the exception to occur, or the elements that become invalid if the action suggested by the modeler is carried out. Application developers use these features to improve modeling support.

The second objective in building the GKOM module was to create a component in a framework for collaborative learning and modeling. In this envisioned framework, which is still in an early stage of development, GKOM-based modeling applications communicate with remote simulation services, with model repositories and with each other. The communication module and the import-export module have been included in GKOM in order to allow GKOM-based applications to operate in this framework, and are currently being used to allow modeling applications to communicate with a remote simulation service. This suggests that it will be possible to add more services to the framework in the future, and confirms that the GKOM knowledge representation is compatible with the GARP knowledge representation: it is possible to transform a GKOM model to a GARP model and vice-versa.

9.2 Discussion

Although GKOM was designed as a component in modeling applications for educational purposes, it could also be used as a component in a modeling tool for expert modelers. However, expert modelers may perceive the GKOM knowledge representation as rather strict. Defining elements at the type-level is required when building a GKOM model, but is not necessary when building a GARP model. Moreover, GKOM-based applications force the modeler to create the type-level elements before creating any occurrence-level elements, which means that modelers must start by defining the entity hierarchy and the properties, relations and quantities at the type-level before model fragments and scenarios can be built. This is acceptable, perhaps even desirable, when GKOM is viewed as a knowledge articulation tool in educational context, but expert modelers using GKOM-based applications as a front-end to the simulator may experience it as a burden. Personal communication with expert GARP modelers suggests that they sometimes start by creating a few central model fragments and subsequently create the entity hierarchy to contain the entities that they found while building the model fragments.

There are two solutions for this issue. One is to allow modelers to loosely define new model ingredients without adding them to the model directly. An example of this is seen in a project carried out by Roland Groen (2003). Groen created a 'sketchpad' in which modelers literally sketch new model pieces. Once the sketch is finished, the modeler can transfer the objects in the sketch to the GKOM model. The second solution is automating the construction of the type-level elements to some degree. Section 9.3 elaborates on that subject.

Another issue is the way in which modeling support is intertwined with the knowledge representation itself. In the GKOM architecture, the support functionality is coded into the knowledge representation module and monitors all modeling activity. Furthermore, the support philosophy is that the modeler is not allowed to do anything that makes a model invalid. But sometimes a modeler may wish to perform a number of actions that – as a sequence – result in a valid model, while one of the individual actions makes the model temporarily invalid. GKOM will not allow this; it will reject the action that makes the model invalid. As an example: it is impossible to replace one entity with another by first removing the old entity and then putting the new entity in its place, because the first step will fail if the old entity has any references to other elements in the model. The result of this strict way of enforcing the validity of a model is that it may prove to be difficult to change an existing GKOM model or to move model ingredients from one model to another.

With the benefit of hindsight, an architecture that separates the knowledge representation from modeling support may have been preferable. In such an architecture, the support system would act as a façade to the knowledge representation, but would allow application builders to bypass the support system and edit the knowledge representation directly. The application builder could then offer functions to the user that map onto a sequence of actions that will succeed even if one of the actions makes the model temporarily invalid.

9.3 Further research

Areas of further research include: the use of GKOM in distributed modeling applications, the use of GKOM in educational modeling applications in which the ontology of the domain is given, and automatic generation of type-level model ingredients.

The framework for distributed modeling and learning, as envisioned in chapter 2, requires two features not currently implemented in GKOM: the ability to communicate with a remote model repository and the ability to exchange fragments of models between modeling applications. A model repository can be as simple as a shared directory on a file server that all computers in a classroom have access to. But a more interesting solution would be a model repository *service*. A repository service could:

- Be made accessible on the Internet, so that learners can share models across a classroom, across schools and even across schools in different countries.
- Host a number of basic models that serve as a starting point for the learner's modeling enterprise.
- Allow learners to store different versions of the same model, thus capturing their increasing understanding of a domain.
- Index models based on properties such as topic, author, or classroom, thus creating model portfolios.

A model repository service would not be part of the GKOM module, but would be a separate piece of software like the simulation service.

Exchanging fragments of models between modeling applications is another interesting topic for further research. Educational modeling applications may allow learners to incorporate pieces of models created by their peers into their own models. Examples include taking a branch of the entity hierarchy of one model and pasting it under a node of the entity hierarchy of another model, or using an existing model fragment or scenario in a new model. To implement this functionality, two research questions must be answered: how can a piece of a model be isolated in such a way that it is self-contained, and how can that piece be inserted into another model. The following example illustrates this subject and deals with how an existing model fragment object from one model could be inserted into another model.

The first step is to create a new model that contains only the model fragment to be copied and all the elements that it refers to. This can be done as follows:

1. Identify the types of all the elements inside the model fragment. This results in a set of EntityTypes, PropertyTypes, RelationTypes, QuantityTypes and QuantitySpaceTypes.
2. For each of the EntityTypes identified in step 1, identify their parent EntityTypes.
3. Create a new model and copy all types identified in steps 1 and 2 to it. The structure of the entity type hierarchy must match that in the original model.
4. Copy the model fragment itself to the new model.

This process would result in a new model, which captures all the knowledge present in the original model about the model fragment that is being copied, and nothing else. The second step is to insert this temporary model in the target model. For this, an algorithm for *merging* models would have to be developed: an algorithm that identifies the similarities and differences between two models and generates a set of issues that the modeler must solve before the 'alien' fragment can be inserted. This process could take advantage of knowledge at the type-level: once the ontologies of the two models are aligned, the model fragment itself can be inserted into the target model. Algorithms for merging ontologies exist, an example being SMART (Noy & Musen, 2000).

The type-level elements in a GKOM model have another potential use: they can serve as a set of building blocks for users to build model fragments and scenarios with. Most of the examples used in this document have considered modeling applications that require learners to build their models from scratch. But one of the desirable features of a modeling application for education, identified in chapter 2, is offering learners a catalogue of model ingredients as a starting point for their modeling exercise. In a GKOM model, the type-level model ingredients can serve this purpose. Using them in this way would require an application that allows a teacher to define a catalogue of elements (in terms of an

entity hierarchy with property types, relation types and quantity types in it) and restricts the learners' possibility of altering that hierarchy. Learners use the elements as a construction kit: dragging an entity type from the hierarchy to a model fragment results in an occurrence of that entity type being added to the model fragment, for example. What makes this an interesting topic for further research is that it is ideal in terms of giving learner support. The support system treats the type-level information as authoritative and in this setup it is. So by defining the catalogue of elements, the teacher is also giving the support system the information it requires to give sensible user feedback.

A final direction for further research is automatic generation of the type-level knowledge in a GKOM model. One of the issues mentioned in the previous section is that GKOM forces a particular way of modeling: the type-level elements must be created before model fragments and scenarios can be built. To allow modelers to build model fragments first and deal with the ontology – in the form of type-level elements – later, a modeling application would have to partially automate the process of creating type-level elements. The simplest way to achieve this is to automatically create a type-level element for an occurrence every time that the modeler does not explicitly assign one. If the modeler creates a new entity, property, relation or quantity, a corresponding type is added to the entity hierarchy. All entity types are added to the hierarchy at level 1, directly under the model's base entity, which is at level 0. All property, relation and quantity types are added to level 0, the base entity itself, so that they are guaranteed to be valid for all entities at level 1. This solution effectively gives the modeler the freedom create model fragments and scenarios first and the ontology later. It also prevents the support system from intervening. But it does not capture any real ontological knowledge about the domain: it simply relates everything to everything.

The research question is to find out to what extent the structure of the entity hierarchy can be inferred from the modeler's actions. For example, it may be possible to determine the entity-inheritance structure from the model fragment-inheritance structure. And with the entity-inheritance structure inferred, it may be possible to put the properties, relations and quantities in their proper place in the hierarchy. Most likely the process cannot be completely automated, but it would be possible to make intelligent guesses. The application can then enter into a dialogue with the modeler, who would make the final decisions.

In this approach, the type-level knowledge plays the role of being an abstraction of the model rather than an authoritative description of the structure of a domain. It can serve as a feedback mechanism. The model is basically asking the modeler: "From what you have built until now, I infer the following domain structure. Is that correct?" In an expert modeling application, this approach may be more appropriate than the authoritative role that the type-level normally plays.

10 Literature:

Bessa Machado, V. 2000. MoBum: a Model Building Environment. Progress report, University of Amsterdam, Amsterdam, The Netherlands.

Bessa Machado, V & B. Bredeweg 2001. Towards Interactive Tools for Constructing Articulate Simulations. Proceedings of the International workshop on Qualitative Reasoning. QR'01, pages 98-104, San Antonio, Texas, USA, May 17-19.

Bouwer, A. & Bredeweg, B. 2001. VisiGarp: Graphical Representation of Qualitative Simulation Models. In: Artificial Intelligence in Education: AI-ED in the Wired and Wireless Future, pages 294-305. Edited by J.D. Moore, G. Luckhardt Redfield, and J.L. Johnson. IOS-Press/Ohmsha, Osaka, Japan.

Bredeweg, B. 1992. Expertise in qualitative prediction of behavior. PhD thesis, University of Amsterdam, Amsterdam, The Netherlands.

Bredeweg, B & R. Winkels 1998. Qualitative models in interactive learning environments: an introduction. In: Interactive Learning Environments volume 5 (1-2), pages 1-18. Guest edited by R. Winkels and B. Bredeweg.

Cañas, A.J., K.M. Ford, P.J. Hayes, J. Brennan & T. Reichherzer. 1995. Knowledge Construction and Sharing in Quorum. Proceedings of the 1995 AI in Education Conference, August 1995, pages 218-225.

De Kleer, J. H. 1990. Qualitative Physics: A personal view. In: Readings in Qualitative Reasoning about Physical Systems, pages 1-8. Edited by D. S. Weld and J. H. de Kleer. Morgan Kaufman, San Mateo, California.

De Kleer, J. H. & J. S. Brown. 1984. A Qualitative Physics Based On Confluences. In: Artificial Intelligence volume 24, pages 1-83. Elsevier Science Publishers, The Netherlands

De Koning, K., Bredeweg, B., Breuker, J. & Wielinga, B. 2000. Model-Based Reasoning About Learner Behavior. In: Artificial Intelligence 117, pages 173-229. Elsevier Science Publishers, The Netherlands

Falkenhainer, B. and K. D. Forbus. 1991. Compositional Modeling: Finding the Right Model for the Job. In: Artificial Intelligence 51 (1-3), pages 95-143. Elsevier Science Publishers, The Netherlands.

Forbus, K. D., P. Whally, J. Everett, L. Ureel, M. Browkowski, J. Baher & S. Kuehne. 1999. CyclePad: An Articulate Virtual Laboratory For Engineering Thermodynamics. In: Artificial Intelligence 114, pages 297-347. Elsevier Science Publishers, The Netherlands.

Forbus, K. D. 1988. Qualitative Physics, Past, Present, and Future. In: Exploring Artificial Intelligence, pages 239-296. Edited by Howard Shrobe. Morgan Kaufman Publishers, Inc.

Forbus, K. D. 2001. Articulate Software for Science and Engineering Education. In: Smart Machines in Education, edited by K. Forbus & P. Feltovich, pages 235-268. AAAI Press, Menlo Parc, California, USA.

Forbus, K., K. Carney, R. Harris & B. Sherin. 2001. A Qualitative Modeling Environment for Middle-school Students: A Progress Report. Northwestern University, Evanston, Illinois, USA.

- Fowler, M. & K. Scott. 2000. UML Distilled: a brief guide to the standard object modeling language. 2nd edition. Addison Wesley Longman, Inc., Reading, Massachusetts, USA.
- Gamma, E. & R. Helm & R. Johnson & J. Vlissides. 1995. Design Patterns: elements of reusable object-oriented software. Addison Wesley, New York, USA.
- Goddijn, F. 2002. Quags - Automatische Vraaggeneratie bij Kwalitatieve Simulaties. Master thesis at the University of Amsterdam. Amsterdam, The Netherlands.
- Groen, R. M. S. 2003. Supporting model building. Master thesis at the University of Amsterdam. Amsterdam, The Netherlands.
- Harold, E & W Scott Means. 2002. XML In A Nutshell, 2nd edition. O'Reilly and Associates, Inc. California, USA.
- Hayes, P. 1978. The naïve physics manifesto. In: Expert systems in the microelectronic age, edited by D. Michie. Edinburgh University Press, Edinburgh, Scotland.
- Jellema, J. 2000. Ontwerpen Voor Ondersteuning: De rol van taakkennis bij ondersteuningsontwerp. Master thesis at the University of Amsterdam, Amsterdam, The Netherlands.
- Kulpa, Z. 1994. Diagrammatic Representation and Reasoning. In: Machine Graphics & Vision 3 (1-2), pages 77-103. Polish Academy of Sciences, Poland.
- Leelawong, K., Y. Wang, G. Biswas, D. Schwartz, N. Vye & J. Bransford. 2001. Qualitative Reasoning techniques to support Learning by Teaching: The Teachable Agents Project. Qualitative Reasoning Workshop, 2001. San Antonio, Texas, USA.
- Novak, J. D. 1977. A Theory of Education Cornell University Press, Ithaca, New York, USA.
- Noy, N. F. & M.A. Musen. 2000. SMART: Automated Ontology for Merging and Alignment. In: Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000), Austin, Texas. Stanford University, Stanford, California, USA.
- Vygotsky, L. S. 1978. Mind in society. Harvard University Press, Cambridge, Massachusetts, USA.
- Russel, S. & P. Norvig. 1995. Artificial Intelligence: A Modern Approach. Prentice-Hall Inc, New Jersey, USA.
- Salles, P & B. Bredeweg. 1997. Building Qualitative Models in Ecology. In: Proceedings of the 11th Qualitative Reasoning Workshop, Italy, June 3-6, 1997
- Satzinger, J. W. & T. U. Ørvik. 1996. The object oriented approach: concepts, modeling and system design. London, International Thomson Publishing.
- Sime, J.A. 1998. Model switching in a learning environment based on multiple models. In: Interactive Learning Environments 5 (1-2) pages 109-124. Guest-edited by R. Winkels and B. Bredeweg. Swets & Zeitlinger, The Netherlands.
- Sime, J.A. & R. R. Leitch. 1992. Multiple models in intelligent training. Proceedings of the Conference on Intelligent Systems Engineering (ISE'92), pages 263-268.
- Sommerville, I. 1995. Software Engineering, fifth edition. Addison Wesley, Essex, England.

J.A. Vadillo, A. Diaz de Ilarraza, I. Fernandez, J. Gutierrez & J.A. Elorriaga. 1998. Behavioural Explanations in Intelligent Tutor Systems for Training using Causal Models. In: Interactive Learning Environments 5 (1-2) 125-134, 1998. Guest-edited by R. Winkels and B. Bredeweg. Swets & Zeitlinger, The Netherlands.