

Collaborative Conceptual Modeling: Share, Search and Reuse

Jochem Liem, Anders Bouwer and Bert Bredeweg

Human Computer Studies Laboratory – University of Amsterdam
Kruislaan 419 (Matrix I), 1098VA Amsterdam, The Netherlands {jliem,bouwer,bredeweg}@science.uva.nl

Abstract

Within the Qualitative Reasoning community there is a desire to collaborate by integrating work and reusing parts of existing models. Although there has been much attention for the knowledge representation formalisms required for these tasks, activities performed by knowledge engineers such as copying model parts, searching for relevant models, and sharing intermediate modeling results are often not supported by existing modeling tools. This paper presents a set of new features in the Garp3 qualitative reasoning and modeling workbench to support these activities.

Introduction

An interesting idea of Qualitative Reasoning (QR) is to build a generic library of model fragments that can be applied by different users to simulate specific scenarios [4,7]. However, in practice the development of unified libraries has been limited. Modelers seem to prefer creating their own idiosyncratic libraries that are tailored to their specific needs, reusing only certain parts of previous modeling efforts, and adapting or leaving out other parts. In any case, whether a modeler wants to integrate modeling work of different modelers to create a unifying library, or reuse existing model fragments to create his own specific library, functionality is required to share and reuse parts of (partially) developed libraries. In this paper we present this functionality as implemented in the Garp3 workbench.

Taking a part of one body of knowledge and integrating it into another has several issues. For instance, knowledge parts usually relate to other knowledge parts. Without this other knowledge semantics may get lost. On the other hand, the existing body of knowledge may clash with the part of knowledge that is being reused. In order to prevent problems arising from such issues, we have defined a set of principles to support knowledge reuse. These are enumerated as: (1) *Syntactical correctness should be maintained*. Knowledge is usually represented in some formalism. After knowledge from one body of knowledge has been added to another, the augmented knowledge body should still adhere to the formalism. (2) *Knowledge should remain complete*. Knowledge parts often depend on other knowledge parts. When knowledge is reused in another body, the knowledge parts on which it depends should also be copied to that new context. (3) *No redundant knowledge should be added*. Two knowledge bodies may have

overlapping parts. When reusing knowledge from one body in another, and some of the knowledge already exists, this knowledge should be reused as much as possible. (4) *Existing knowledge should not be altered*. The knowledge a modeler is working on can be assumed to be tailored to the needs of this modeler. Therefore, the knowledge should not be changed when knowledge from another knowledge body is added, as it could break the purpose for which the knowledge was developed. (5) *Semantics should be preserved as much as possible*. Copying knowledge should not cause the meaning of the knowledge to be changed or lost. (6) *Reuse solutions should be user-friendly*. This means the modeler should not have to provide too much additional input, be asked difficult questions, and that the functionality is easy to use.

Next to reuse functionality, two other conditions have to be fulfilled to efficiently reuse previously created bodies of knowledge. Firstly, modelers should be able to share their work within a community and the shared knowledge should be made searchable and accessible to the entire community. Otherwise, there is no knowledge to reuse. Secondly, modelers should be able to search through the shared knowledge in order to find knowledge that is potentially reusable for their needs. In the Garp3 workbench search is facilitated by meta-data, the model itself, and by high-level descriptions of the model and its expected behavior, referred to as Sketches.

Garp3 – QR Workbench

In this paper, the reuse functionality is addressed in the context of the Garp3 workbench (<http://www.garp3.org>), which implements a diagrammatic approach to modeling and simulating qualitative models [2]. Modeling in Garp3 starts by creating *model ingredient definitions*. These definitions include entities, agents, assumptions, configurations, quantities and quantity spaces. *Entities*, which represent the structural objects in a system, are organized in a sub-type hierarchy. They are defined by their name and their position in the hierarchy. Agents and assumptions are defined in the same way. *Agents* cause influences from outside of the system, while *assumptions* are labels that indicate that certain conditions are presumed to be true. *Configurations* are structural relations between entities that are defined by their name. *Quantities* represent the features of entities and agents that change during

simulation, and are defined by their name and a set of possible quantity spaces. *Quantity spaces* represent the possible values a magnitude (or derivative) of a quantity can have, and are defined by their name and an ordered set of possible values. Quantity spaces are associated to the quantities of entities or agents.

Next to the model ingredients defined by the modeler, there is also a set of predefined model ingredients. These include causal dependencies (proportionalities and influences), correspondences, the operator relations plus and minus, value assignments, and inequalities.

The model ingredient definitions described above can be used (instantiated) to create model fragments (MFs) and scenarios. MFs can be seen as composite ingredients that incorporate other ingredients as either conditions or consequences. They are organized in a subtype hierarchy, meaning that a child MF inherits the model ingredients of its parents. Furthermore, a MF can incorporate other MFs as conditional ingredients. MFs instantiated in another MF are called Imported Model Fragments (IMFs). An example MF incorporating another MF twice is shown in Figure 1.

Scenarios are also composite model ingredients. They describe specific system situations. During simulation, MFs are sought which match on the scenario (i.e. the model ingredients fulfill the conditions of the MF). The consequences of matching MFs are merged with the scenario to create an augmented state from which the next states of behavior can be determined.

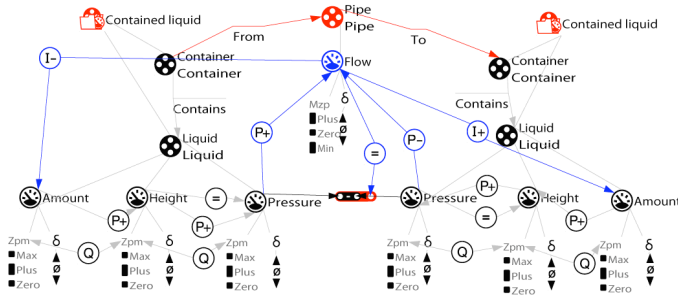


Figure 1: Liquid flow includes two Contained Liquid IMFs, the Pipe, and the configurations as conditions, and flow, its calculation and causal relations as consequences.

Reusing Parts of Models

A user-friendly way to support modelers with functionality to reuse model parts (model ingredient definitions, MFs and scenarios) is by allowing them to copy a model part in one model and paste it into another model (hereafter called *copy functionality*). From the modeler's perspective, copying model parts should be as easy as copying text between documents (principle 6).

The result of a copy should be assured to result in a syntactically correct model (principle 2). For example, ingredients must have unique names and arguments of relations must have correct type. This is achieved by rebuilding the copied model part in the target model as if the modeler had created it from scratch. Since Garp3

checks each of the modeler's actions, this assures that the model remains syntactically correct.

When a model part is copied to another model, no redundant information should be added to the model (principle 3). Therefore, when a model part and an already existing model part have the same name, the existing model part is reused if possible. The assumption is that if the copied model part and the existing model part have the same user-given name, they describe concepts in the same domain. If the existing model parts cannot be reused because semantics differ despite having the same name, the copied model part receives a suffix to indicate this.

Storing Copied Model Ingredients

Model parts often depend on other model parts without which they are incomplete. The completeness of the model part has to be maintained (principle 2) when a model part is copied. Our solution is to create a complete sub-model that contains the copied model part and all the model parts it requires. This sub-model is self-contained, meaning it can exist on its own, and is stored in a copy buffer (a model data structure). Details on how this works for each type of model part are explained in the next subsections.

Copying Model Ingredient Definitions

Entities are defined by their name and their position in the entity hierarchy. When a set of entities is copied to another model, they have to be integrated with the already existing entity hierarchy in some way. If an entity with the name already exists, redundancy should be avoided (principle 3). Therefore, the entity is not created, since it assumed to represent the same concept.

The entities should be integrated into the hierarchy in such a way that as much of the semantics is preserved as possible (principle 5). For entities this means that their position in the hierarchy should match as closely as possible. This is not straightforward since the modeler can select a subset of the entities in different branches. Only these selected entities should be copied, as it would not be user-friendly if the copy functionality would add more entities than the modeler has selected (principle 6). As a result, the final entity hierarchy will not always contain the parents of each entity.

Our solution to preserve as much of the semantics as possible is to also store all the ancestors of the selected entities in the copy buffer. When the selected entities are copied to the target hierarchy, the algorithm checks whether an ancestor (parent, grandparent, etc.) of each entity already exists. The entity is placed below the closest ancestor to recreate its semantics as closely as possible. If, no ancestor exists, the definition is placed below the root node. The modeler is allowed to choose a different position in the hierarchy where the copied entities should be recreated (principle 6). The hierarchy of entities is then created below this definition. Agents and assumptions are defined and copied in the same way.

Configurations are the simplest model ingredients, since they are only defined by their name. To avoid redundancy (principle 3), a copied configuration is only created if it does not exist yet in the target model.

Quantity spaces are defined by their name and their total order of values. The order of the values is important, since it defines to which values the magnitude (or derivative) of a quantity can change. Again, redundancy has to be avoided (principle 3). Therefore, a quantity space is not created if a quantity space with the same name and the same set of ordered values already exists. When a quantity space with the same name does not exist, it can be created normally. However, if a quantity space with the same name but with different values (or differently ordered values) exists, the values of the existing quantity space cannot be altered to match the values of the copied quantity space. The reason is that changing an existing quantity space would potentially alter the possible values of already existing quantities and cause simulations to generate different behavior (violating principle 4). Instead, a new definition is created with the suffix '(other values)'.

Quantities are defined by their name and a set of associated quantity spaces (of which only one can be chosen when it is added to a MF). To assure completeness (principle 2), the associated quantity spaces have to exist before the quantity can be created. Therefore, the associated quantity spaces of a quantity are created (as described above) before the quantity is copied. A quantity is copied normally if a quantity with the same name does not exist. A quantity is considered redundant and is not created if a quantity with the same name and the same quantity spaces already exists (principle 3).

When a quantity with the same name already exists, but has different associated quantity spaces, there are two options. The first option is creating a new quantity by adding the suffix '(different quantity spaces)' to its name. This potentially adds redundant knowledge (violating principle 3). The second option is to merge the sets of associated quantity spaces, which means existing knowledge is altered (violating principle 4). We choose this second option in our approach, since the associated quantity spaces only indicate the *possible* values for quantities. Therefore, adding quantity spaces to the set of possible quantity spaces does not really change the semantics of the quantity in model fragments (i.e. the simulation results remain the same), but only provides the possibility to use the quantity in a different way.

Copying Model Fragments

To copy a MF to another model, the algorithm has to deal with the subtype hierarchy, the MFs imported as conditional elements (see Figure 1), the model ingredient definitions of which instances are used in the MF, and the actual creation of the MF and its contents.

Dealing with imported and parent model fragments.

In order to create a MF, all MFs it imports and its parent MFs have to exist. Each parent MF and reused MF has the same requirements. Therefore copying a set of MFs

requires that their required MFs are collected and created first. A list of the to-be-created MFs and their required MFs is determined in several steps. Firstly, MFs inherited from parents are considered to be IMFs. Secondly, the IMFs within the MFs are gathered. Thirdly, the MFs corresponding to these IMFs are added to the list of required MFs. For each of the MFs added to the list the same three steps are performed until no more IMFs can be found. Finally, duplicates in the list of required MFs are removed, and the to-be-copied MFs are added to the list.

Determining and copying used model ingredient definitions. In addition to the model ingredients that the copied MFs use, also the model ingredients that the required MFs use have to be created. Given the MFs list created in the previous step, finding the required model ingredient definitions is easy. A list of model ingredient definitions is created for each model ingredient type. Then, by looping through the model ingredients of each MF, the definition of each model ingredient is added to the list of its type (if it is not already there). The end result is a set of lists that contain all model ingredient definitions needed to create the set of MFs.

The required definitions are copied as if the definitions were individually copied (as described in the 'Copying Model Ingredient Definitions' section). There is one difference when dealing with entities, agents and assumptions. During development it became apparent that the semantics about entities was lost when certain scenarios (see Section 'Garp3 – QR Workbench') are copied before MFs. Scenarios tend to use more specific concepts (lower in the hierarchy) since they represent specific situations, while MFs use general concepts (higher in the hierarchy) since they model general situations. Consider a model that defines the entities container, barrel (which is a type of container), liquid, and water (a type of liquid). Copying a scenario modeling a barrel with water would add both the barrel and the water concept to the hierarchy in the target model. Copying a MF that models a liquid in a container afterwards, would add the container and the liquid to two other branches in the hierarchy, since no information about the children of entities is stored in the copy buffer. The fact that barrel is a container, and that water is a liquid would be lost. This issue is solved by not only copying the required definitions, but also their ancestors. In the example, the liquid and container concepts are immediately created when the scenario is copied, preserving the semantics (principle 5).

Creating the model fragments and their contents.

After these steps, all required model ingredient definitions are present, and each of the MFs in the list (of required and to-be-copied MFs) has to be created. The parents and the MFs each MF imports have to exist before that MF can be created. Therefore, the order in which the MFs are created is important. Instead of ordering the MFs, the algorithm loops through the list of MFs and checks whether the required MFs exist for each MF. If not, it skips to the next one. If they exist, the MF is created and the MF is removed from the list. This continues until the list is empty.

The creation of a MF also requires the creation of the contents of the MF. Again, the order in which the model ingredients are created is important, since Garp3 prevents model ingredients to be created that would result in a syntactically incorrect model (principle 1). Therefore, relations cannot be created if their arguments do not exist (e.g. a ‘preys on’ relation between two populations cannot exist without the two populations), quantities need their entities (e.g. a ‘size’ quantity of a ‘population’ cannot exist without the population), and value assignments need a quantity space before they can be created. To create the MF contents while maintaining a syntactically correct model at all times, the ingredients in the source MF are ordered. IMFs have to be created first, as model ingredients can be related to one of the model ingredients in the IMFs. Then, entities, assumptions, quantities, attributes, configurations, causal dependencies, value assignments, correspondences, plus or minus relations, and inequalities should be created. The algorithm loops through the sorted list of model ingredients creating each of the model ingredients and assigning it the same position on screen as in the source model.

Imported Model Fragments

When recreating the model ingredients of a MF in another model, it is complex and inefficient to have to determine for each model ingredient to which other model ingredient it is connected (e.g. to which entity a quantity should be added, or what the arguments of a relation are). The model ingredient(s) to which a model ingredient is related are called its *arguments*. To be able to determine the arguments of a to-be-created model ingredient in a target MF, a mapping has to be maintained between the ingredients in the source MF and the ingredients in the target MF. The model ingredients in the target MF to which the arguments in the source MF are mapped are the arguments of the to-be-created model ingredient.

Maintaining a mapping between the model ingredients in a source MF and a target MF is easy when model ingredients are created one at a time. When looping through the ordered list of model ingredients in the source MF to create the model ingredients in the target MF, the model ingredient in the list is mapped to the newly created ingredient. However, when a MF is imported, a set of model ingredients is added to the MF. Therefore, a mapping between the model ingredients of the IMF in the source model and the ingredients of the IMF in the target model is harder to establish.

To create the mapping, the model ingredients are sorted in the same way as when creating the contents of a MF. The algorithm loops through the sorted model ingredients of the IMF in the source MF, and selects one of the imported model ingredients of that type in the target model. It checks whether the ingredient has the same name, associated arguments and relations. If the checks succeed, the correct model ingredient is chosen and a mapping between the ingredient in the IMF in the source MF and the ingredient in the IMF in the target MF is

saved. Creating this mapping for each of the model ingredients in the IMF always succeeds, since the IMFs are guaranteed to be identical in both models. The mapping is also used to update the positioning information for the model ingredients in the IMF in the target MF.

Reusing Existing Model Fragments

When a set of MFs is copied to another model, they might clash with MFs that already exist in the target model. These MFs cannot simply be reused, since the semantics of these MFs might be different. The existing MF might contain different model ingredients, or model ingredients might be differently connected than the model ingredients in the source MF. To avoid redundancy (principle 3), the existing MF should be reused if possible. A MF can only be reused if its ingredients are equal or a superset of the ingredients in the source MF, and if the corresponding model ingredients are connected in exactly the same way.

To determine if a MF can be reused, the mapping algorithms used to deal with IMFs is used. The source MF and the target MF are treated as IMFs, and the algorithm tries to create a mapping between the contents of the MFs. This mapping only succeeds if the target MF contains at least all the model ingredients that are in the source MF, and the model ingredients are connected in the same way. In contrast to the mapping between IMFs, the mapping can also fail, meaning that there is no mapping possible and that the MF cannot be reused. Then, the semantics of the to-be-created MF is different from the existing MF, and the MF is created with the suffix ‘(copy)’. When the MF is reused in other copied MFs, this new copy is used instead of the existing MF to preserve the semantics of the copied model fragment (principle 5). The existing MFs keep using the existing MF (preserving principle 4).

To preserve the semantics of a MF (principle 5), the reused MFs should be identical to those in the source model, as reusing different MFs results in a different complete MF. On the other hand, not reusing a MF which is a superset of the source MF (i.e. contains more ingredients), but which is otherwise equal requires a new, possibly redundant, MF to be created (violating principle 3). We feel that the best way to solve this issue is to ask the modeler for feedback. Although this is a difficult question, it makes the modeler aware that there are options, and each choice has a significantly different end result. This solution is more user-friendly than letting the algorithm make the choice for the modeler (principle 6).

Sharing and Searching for Models

To reuse models of others, modelers have to be able to share their work and access work of others. This is solved by allowing models to be uploaded to and downloaded from a central online model repository. However, the number of models in the repository can potentially become large, which means that modelers need to be supported by search functionality to find reusable models.

Typically, a modeler will want to search for models which contain a certain entity or quantity (e.g. a model which contains both an entity *population* and a quantity *size*). Normal search engines search for keywords in text and are unable to interpret the explicit knowledge representation in qualitative models. So the search engine is unable to distinguish between different types of model ingredients, or between domain specific and domain independent knowledge (i.e. the QR vocabulary and the knowledge formalized by the modeler). This hampers the search engine's ability to find relevant models. A search solution should make use of the explicit knowledge representation in qualitative models to allow modelers to focus their search using the QR vocabulary.

QR Models in the Web Ontology Language

The Semantic Web initiative proposes that "semantic search" becomes possible by making content machine-accessible [1]. The Web Ontology Language (OWL) is a description-logic based knowledge representation language, which is represented in RDF/XML, and is being developed as part of the Semantic Web initiative. It has become the de-facto standard for the sharing of knowledge on the web in the form of *ontologies*. By formalizing qualitative models as OWL ontologies, the models become interpretable by OWL search engines, and searching for models in which certain model ingredients or certain structures are used becomes possible. Additionally, the formalization of models in OWL opens up the possibility for other QR-tool developers to implement functionality to import these files. This could potentially make models accessible to communities using different QR tools.

There is no clear methodology for the creation of ontologies, therefore we have created our own. Firstly, the qualitative reasoning vocabulary was formalized as an ontology that consists of a hierarchy of all the model ingredients and their usage restrictions. Based on this domain-independent ontology, an OWL file-format for qualitative models was developed that refers to concepts defined in the vocabulary ontology. Using OWL reasoners, both the vocabulary and a set of model ontologies were checked for consistency, and the model ontologies were checked for correctness using the formalized usage restrictions. Functionality to export models to OWL and import them again was added to Garp3. The machine-accessible OWL-model files allow search engines to use the explicit knowledge representation of QR models.

Originally, we had the aim to use OWL reasoners to perform QR reasoning, but this proved to be impossible. Since the OWL reasoners are classification engines, the formalization should allow scenarios to be classified as being instances of MFs. However, due to limits in the expressiveness of OWL it is not possible to formalize MFs in a way that this reasoning can be performed [13]. In general, it is impossible to formalize general situations in OWL in a way that specific situations can be classified [11]. Due of this lack of expressiveness, the OWL

representation of MFs needed to be adapted. However, this change has little effect on model search.

An earlier effort to support the interchange and reuse of MFs is the Compositional Modeling Language (CML) [5], which aimed to enable this functionality by defining CML in the Knowledge Interchange Format [10]. We have chosen to use OWL instead of CML, since it has a large user base and tools that are being actively developed.

Sharing and Searching in the Model Repository

A qualitative model repository¹ was implemented as a webpage that allows modelers to share their own models as OWL files, and search and download models of others. The main issue of implementing the repository is making it usable for modelers. The repository should be instantly usable for the user. Therefore, modelers should not be required to learn an OWL query language.

There are two different ways of implementing search functionality. The first is building an interface on top of an OWL query language, and the second is programming our own solution. Since building an interface on top of an OWL query language is complex, and implementing dedicated solutions has become easier due to the availability of semantic web libraries, we have chosen the second solution. The model repository is developed using the SWI-Prolog Semantic Web Library² and PHP³.

The search functionality shows the model ingredient definitions of all the models. Selecting a definition reduces the list of matching models, allowing the modeler to iteratively refine the list of potentially useful models.

Sketch: Supporting Structured Modeling

The Garp3 workbench has been extended with the Sketch environment to allow modelers to create high-level representations of systems before starting the model implementation. The goal of the Sketch environment is threefold:

- to offer guidance during the modeling process, by providing editors that support different steps in the structured modeling methodology [3];
- to document initial ideas and intermediate modeling decisions by allowing the creation of external representations for them. Although not all captured ideas may end up in the final model, these Sketch representations can aid in communicating about the domain and establishing consensus between collaborating modellers;

¹ <http://hcs.science.uva.nl/QRm/models/repository/>

² <http://www.swi-prolog.org/packages/semweb.html>

³ <http://www.php.net>

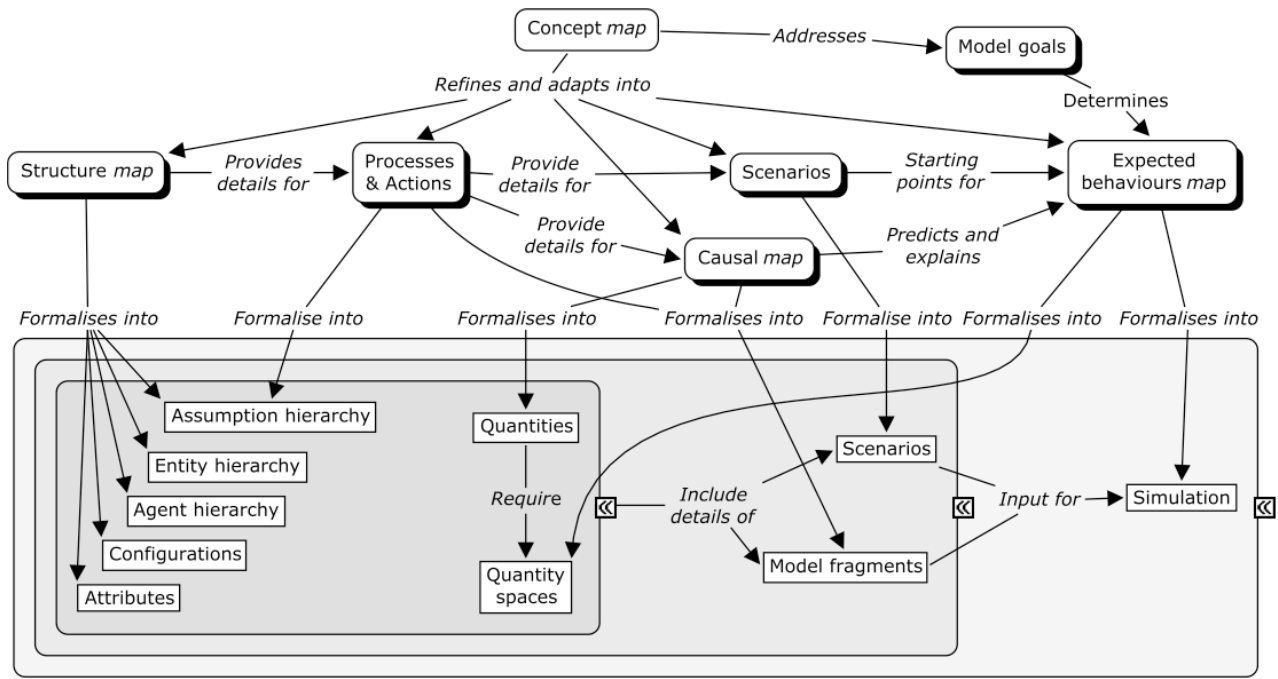


Figure 2: Overview of the intermediate representations used in the structured approach.

- to facilitate determining whether an existing model is relevant for a modeler, by providing a set of Sketch representations as a high-level abstraction and introduction to the model. Together with the metadata that was already introduced in Garp3 [2] (including abstract, keywords, and descriptions of the model goals, domain, and intended audience), this makes it possible to find out what the model is about, without having to analyze the details of the model implementation, which might be hard to understand at first glance.

Compared to the Build environment interface of Garp3 as described in [2], the editors in the Sketch environment have been designed to have a sparser user-interface. Each editor focuses on a specific kind of knowledge, so that the modeler has to focus on only a few types of ingredients per editor. Furthermore, the Sketch editors do not impose some of the grammatical constraints associated with the model implementation (e.g., quantities do not have to be associated to an entity, and quantity space values do not have to be characterized as points or intervals), to facilitate the flow of ideas in the initial stages of modeling. Not enforcing these constraints does not create a problem in the Sketch environment because the Sketches are not used directly as input for the simulation engine.

The Sketch Editors

The Sketch⁴ environment consists of seven different editors. Their recommended use is in the order matching Figure 2, which shows an overview of the intermediate modeling results and how they follow up on and refine each other.

⁴ The term ‘Sketch’ is used here to refer to the preliminary and relatively unconstrained nature of the representations, rather than free-form drawing

In the Concept Map editor, inspired by the IHMC Cmap Tools [14], a modeler specifies the concepts and relationships that are considered important in the domain as a graph consisting of labeled nodes and links, respectively. No additional building blocks or constraints are given at this stage (such as having to create modeling ingredients in a particular order), allowing the modeler to freely specify his or her initial ideas. The concept map addresses the model goals and serves as a basis for refinement into the other Sketches.

In the Structural Model editor, the modeler needs to focus on the physical structure of the system and how it relates to the environment. The graphical format is similar to the Concept Map editor, but here each node is assigned a type (entity, agent, assumption, or undefined concept). This guides the modeler to be more specific about the nature of what is represented. Common structural relationships have been predefined (connected-to, contains, is-a), but the modeler can add new relation definitions as well.

The generic knowledge about system behavior can be represented in three editors: the Process Definitions editor, the Actions and External Influences Definitions editor, and the Causal Model editor. The Process Definitions editor allows the modeler to define processes that affect the system by specifying the related entities, quantities, start conditions, effects, stop conditions, and behavioural assumptions. The Actions and External Influences Definitions editor is used to specify influences exerted from outside the system, and is similar to the Process Definitions editor except for an additional field for the agents causing the influence. The Causal Model editor is used to describe the causal dependencies between quantities, to indicate how they affect each other. This type of editor relates to tools such as VModel [9] and Betty’s Brain [12]. In the Sketch Causal Model editor there are

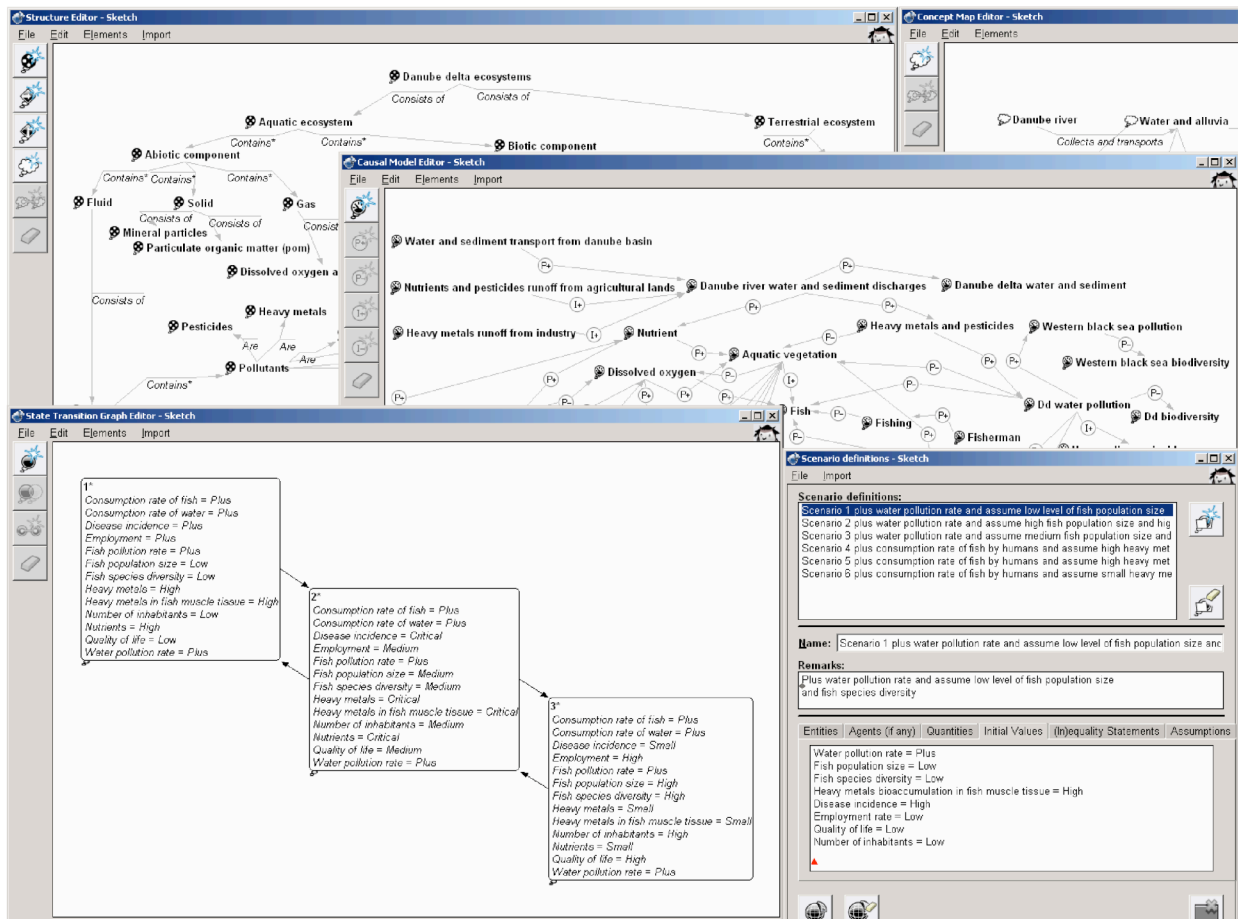


Figure 3: A screenshot of the Sketch environment.

four types of causal relationships: they are either direct or indirect, and either positive or negative [8]. Together, they provide an overview of the effects of the processes and actions defined in the previous two editors, and how these effects propagate through the system.

Finally, there are two editors that deal with specific behavior: the Scenario Definitions editor, and the Behavior Graph (or Expected Behaviors Map) editor. In the Scenario Definitions editor, scenarios can be specified to represent different initial situations of the system, which will be the starting points in the system's behavior. In this structured text based editor scenarios can be defined by specifying the entities, agents, quantities, initial values, (in)equality statements, and behavioral assumptions that pinpoint what is relevant in determining the behavior of the system.

In the Behavior Graph editor, the modeler can indicate how quantities and (in)equalities are expected to change over time given an initial scenario. The modeler creates the states, defined by a set of value and (in)equality statements, and possible transitions between them to represent the main aspects of the system's anticipated behavior. The value and (in)equality statements are displayed within the state nodes, to present a clear overview of the content of the possible behaviors. A screenshot of several of the Sketch editors is shown in Figure 3. The contents of the figure are taken from a case

study within the NaturNet-Redime project about the Danube Delta Biosphere Reserve [6].

To further support working through the structured modeling methodology (following Figure 2), it is possible to import certain parts from one Sketch into another, thereby enabling reuse and refinement of ideas. For example, concepts specified in the Concept Map editor can be imported (and refined into other types) in the other editors. Entities, agents, and assumptions specified in the Structural Model editor can be imported in the Process Definitions editor, the Actions and External Influences Definitions editor, and the Scenario Definitions editor.

Conclusions and Future Work

This paper presents new collaborative modeling features of the Garp3 qualitative reasoning and modeling workbench [2] to further facilitate the articulation of knowledge. Engineers of conceptual knowledge use Garp3 to construct qualitative models. Particularly, partners in the NaturNet-Redime project use the workbench to capture knowledge about river restoration ecology.

To prevent redoing of work within a community a central online model repository has been developed in which qualitative models (formalized in the Web Ontology Language) can be shared and searched for. Within Garp3 multiple model support and copy functionality have been

added so that model parts can be easily reused. This makes it possible to reuse parts of existing models, integrate models to create larger models, and create alternative representations of systems to share within communities.

To support synchronous collaborative modeling the Sketch environment has been developed. Sketch helps consensus building through explicit representations to focus discussions and solidify established consensus. Another role of the Sketch environment is to ease the transition from initial ideas to implementation of the model, following a structured approach to model building [3]. Because the Sketches provide a high-level description of the implemented model, inspecting the Sketches can also help modelers to determine if a particular model is useful for them, without having to inspect the details of the model implementation itself. This is another added value.

Future work will focus on three issues. First, using the Sketch representations to (partially) automate model construction. Because the representations used in the Sketch environment are less constrained than the definitive Garp3 format for model implementation, certain model ingredients from the Sketches (e.g., the structural model, the causal model, the processes, and scenarios) might be reused or refined into the final model. The State-Transition Graph Sketch that represents anticipated behaviors can be compared to the actual simulation results to find discrepancies that may be used to refine the model, or the expectations.

Second, reusing model parts can cause undesired behavior during simulation. Investigating what kinds of issues occur, and how results deviate from modeler's expectations will further the design of repair methods, and eventually, automated support for troubleshooting.

Third, studies with modelers are planned, in the context of the NaturNet-Redime project, to evaluate the new functionality.

Acknowledgements

The research presented here is co-funded by the European Commission within the 6th Framework Programme for Research and Development (2002-2006) (project NaturNet-Redime, number 004074, www.naturnet.org). We thank the reviewers and the participants of the NaturNet-Redime workshops in Sofia, Bulgaria (March 2006) and Birini, Latvia (September 2006) for their insightful feedback.

References

- [1] Antoniou, G. and van Harmelen, F. A Semantic Web Primer, The MIT Press, Cambridge, Massachusetts, April 2004.
- [2] Bredeweg, B., Bouwer, A., Jellema, J., Bertels, D., Linnebank, F. and Liem, J. Garp3 - A new Workbench for Qualitative Reasoning and Modelling. 20th International Workshop on Qualitative Reasoning (QR-06), C. Bailey-Kellogg and B. Kuipers (eds), pages 21-28, Hanover, New Hampshire, USA, 10-12 July, 2006.
- [3] Bredeweg, B., Salles, P., Bouwer, A., Liem, J., Nuttle, T., Cioaca, E., Nakova, E., Noble, R., Caldas, A.L.R., Uzunov, Y., Varadinova, E. and Zitek, A. Towards a Structured Approach to Building Qualitative Reasoning Models and Simulations. *Ecological Informatics* (in press).
- [4] Bredeweg, B. and Struss, P. (eds). 2003. Current Topics in Qualitative Reasoning. *AI Magazine (special issue)*, Volume 24, Number 4 (winter), pages 13-130.
- [5] Bobrow, D., Falkenhainer, B., Farquhar, A., Fikes, R., Forbus, K., Gruber, T., Iwasaki, Y., and Kuipers, B. (1996). A Compositional Modeling Language. In Iwasaki, Y., and Farquhar, A., editors, *Proceedings of the Tenth International Workshop for Qualitative Reasoning (QR-96)*, pages 12-21, AAAI Press, AAAI Technical Report WS-96-01, Menlo Park, California, USA.
- [6] Cioaca, E., S. Covaliov, C. David, M. Tudor, L. Torok, O. Ibram, Textual description of the Danube Delta Biosphere Reserve case study focusing on basic biological, physical, and chemical processes related to the environment. Naturnet-Redime, STREP project co-funded by the European Commission within the Sixth Framework Programme (2002-2006), Project no. 004074, Project Deliverable Milestone M6.2.1.
- [7] Falkenhainer, B. C. and K. D. Forbus. Compositional modeling: Finding the right model for the job. *Artificial Intelligence*, 51:95-143, 1991.
- [8] Forbus, Kenneth D., Qualitative Process Theory, *Artificial Intelligence*, 24, p. 86-168, 1984 .
- [9] Forbus, K., L. Ureel, Carney, K., and Sherin, B.. Qualitative modeling for middle-school students. In J. de Kleer and K. D. Forbus (eds.) *Proceedings of QR 2004, 18th International Workshop on Qualitative Reasoning*, Evanston, USA, August 2-4, 2004, pp. 81-88, 2004.
- [10] Genesereth, M. R. and Fikes, R. E. (1992). Knowledge Interchange Format, Version 3.0 Reference Manual. Technical Report Logic-92-1, Stanford University Logic Group.
- [11] Hoekstra, R., Liem, J., Bredeweg, B. and Breuker, J. Requirements for Representing Situations. *Proceedings of the OWLED'06 workshop on OWL: Experiences and Directions*. In Bernardo Cuenca Grau and Pascal Hitzler and Conor Shankey and Evan Wallace (Eds.): *CEUR Workshop Proceedings 216*. Athens, Georgia, USA, November 10-11 2006.
- [12] Leelawong, K., Y. Wang, G. Biswas, N. Vye, J. Bransford, and D. Schwartz. Qualitative reasoning techniques to support learning by teaching. In G. Biswas, (ed.), *Proceedings of QR 2001, 15th International Workshop on Qualitative Reasoning*, St. Mary's University, San Antonio, Texas, 17-18 May 2001, pp. 65-72, Stoughton, WI, 2001. The Printing House.
- [13] Liem, J. and Bredeweg, B. OWL and qualitative reasoning models. In C. Freksa, M. Kohlhase, and K. Schill (Eds.): *KI 2006, Lecture Notes in Artificial Intelligence 4314*, pp. 33-48. Springer-Verlag Berlin Heidelberg 2007. (to appear)
- [14] Novak, J. D. and D. B. Gowin. *Learning How to Learn*. Cambridge University Press, New York, 1984.