Collaborative Modeling in Graph Based Environments

Vom Fachbereich Ingenieurwissenschaften der
Universität Duisburg-Essen

zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

genehmigte
Dissertation
von

Niels Pinkwart
aus Orsoy (jetzt Rheinberg)

Referent: Prof. Dr. H. Ulrich Hoppe
Korreferent: Prof. Dr. Daniel D. Suthers
Tag der mündlichen Prüfung: 19. Juli 2005

Acknowledgements

There are a number of people who have helped and encouraged me in completing this work.

Foremost, I would like to thank Ulrich Hoppe, the advisor of this thesis, for motivating and encouraging me, and for the support he provided. I am grateful for the chance of being a member of his research group during the last years of preparing and writing this thesis. Also, I would like to thank Dan Suthers for co-advising this work. The "hawaiian style" experiences that I could gather during my visit in his lab allowed me to view my work from a helpful additional perspective.

I thank the current and former members of the COLLIDE research group. The creative and inspiring working atmosphere in this group has been a continuous source of inspiration for this work. In addition, the usage of the different Cool Modes prototypes during the last years and the resulting feedback pushed forward the development. I am thankful to Katrin Gaßner and Andreas Lingnau, who have shared an office with me for years, for their collaborative working style and the pleasant working environment. Most of the research and student assistants of the COLLIDE group have provided important assistance in programming tasks, and have given valuable contributions during countless technical discussions. I am also grateful to Sam Zeini for advising me in evaluation methodology issues, and to the teachers and programmers who volunteered for the interviews. Without their collaboration, the evaluation parts of this thesis would not have been possible.

A special thanks goes to Andi Harrer. He has spent hours proof-reading this thesis and discussing it with me. Such a kind of support is invaluable.

Finally, I want to thank my wife Julia. Her patience and understanding made this work possible. I love her.

Abstract

This thesis presents a conceptual approach and a corresponding system implementation to support collaborative modeling with graph based representations. Motivated by promising application potential in the field of education, the work is methodologically rooted in computer science, at the intersection of fields like metamodeling, software design, distributed systems, and visual languages in HCI contexts. In this combination of computer science methods applied with a view towards educational scenarios, the work is in the tradition of research fields like groupware, AIED, and CSCL.

In introductory requirements analysis parts, different notions of interoperability (syntactic, semantic, social, and task) and further criteria for a collaborative modeling system relying on graphs as primary representations are worked out. Reviews of the current state-of-art in dynamic and collaborative modeling tools and theory show that there are neither existing software environments that meet these requirements, nor conceptual approaches that address all the criteria.

Taking up concepts and approaches from graph theory, visual language theory, and metamodeling, this thesis presents the concepts of visual typed graphs and Reference Frames as formal notations for models and modeling languages usable in collaborative contexts. On this conceptual level, interoperability issues for shared heterogeneous models are discussed.

Building upon existing libraries for communication support and graph representations (identified through a requirement list which takes into account the needs of the abstract Reference Frame concept), an architecture as a base for system implementations of the Reference Frame approach is presented. This architecture offers rule based syntax specification options and support for model synchronization. Furthermore, it contains a lightweight event based mechanism for model interpretation (which comprises model simulation functionality), and options for Reference Frame interoperability also on the implementation level - the latter corresponding to relations already exemplified on the conceptual level. Central design decisions, including trade-offs between expressiveness and flexibility on the one hand, and central control on the other hand, are discussed.

The Cool Modes modeling framework as one example implementation of the Reference Frame architecture is presented in detail. This application manages multiple Reference Frames and supports collaboration relying on a "shared workspace" metaphor. Through specific user interfaces for Reference Frames (called Palettes), the users are provided with an easy means to build heterogeneous models using primitives (nodes and edges) that (externally definable and "pluggable") modeling languages provide. Cool Modes as a tool which offers multiple work phases and several forms of partial synchronization (relying on language-specific synchronization contexts) is described.

This thesis concludes with an evaluation of the Cool Modes system: several non-technical requirements which are not suitable for formal proofs (like, e.g., social interoperability) are evaluated on the base of interviews conducted with programmers who have used the application framework, and with teachers who have conducted lessons using the Cool Modes environment.

Contents

Chapter 1

Introduction

The ability to learn is absolutely essential for human beings in order to acquire knowledge, skills, and behavior and thus to adapt to a changing reality. Consequently, questions dealing with the factors that determine human learning have been an important subject of research in the past decades. Researchers have not only been exploring the nature of learning in various general perspectives, but have in particular formulated a number of approaches that deal with specific conditions in which learning may occur. Some of these approaches are more from a practitioner's point of view, others base on a scientifically sound set of principles. According to Schunk (1991), an important role of the latter type - learning theories - is that they

> *"[...] provide frameworks for making sense of environmental observations, serving as bridges between research and educational practices and as tools to organize and translate findings into recommendations for educational practice."* (page 17&18)

Gredler (1992) identifies four functions that theories of learning have in the different educational disciplines:

- Serve as guidelines to design instruction
- Allow for evaluation of current products and practices for classroom use
- Diagnose problems in the educational situation
- Evaluate research conducted on theories

In all these different functions, theories of learning are concerned with factors that determine or influence learning. While for quite a long time, these factors mainly played a role in "traditional" educational settings (like classroom situations), evolving new possibilities available with developing computer technology have opened some new directions. Starting with Computer Assisted Learning approaches that came up in the 1960s (Suppes & Macken, 1978), the possibilities that technology as a means for learning support offers became a subject of research. Typical questions in this intermediate area between pedagogy and computer science deal with the design and use of software tools that support learning.

Not all the tools built in this area have been explicitly designed in context of a specific learning theory. Even if they are, an important factor to understand is that the theoretical foundations are mostly far from being operational enough to allow for a direct and determined derivation of a learning environment. In any case, it is obvious that, if a learning theory is adopted and taken as premise for designing computer-based learning tools, this theory has an impact on the properties

and foci that the resulting tools have. For instance, building upon behavioristic positions like presented by Skinner (1974) will almost definitely result in systems that implement stimulus-response patterns and offer reinforcements or punishments. Another example: a foundation on social learning theory like presented by Bandura (1977) is likely to produce systems in which learners can observe other (real or artificial) agents' actions and their consequences.

Of course, the described impact is not direct. Given this fuzzy but existing relation and the heterogeneity of existing learning theories, it is not surprising that *the one* educational piece of software that supports learning in all its (theoretical and practical) varieties does not exist. In addition, we can not observe tendencies towards a unifying learning theory, which could be used to derive design guidelines for general purpose educational software tools. Even if such a theory existed, it is unclear in how far it would or could have operational characteristics that could directly and deterministically guide implementations, in particular taking into account the complexity of learning scenarios and, therefore, the need to drop "closed world/full control" positions (Hoppe, 2005, to appear).

Despite these points of criticism (fuzzyness, heterogeneity and missing operational character), there is a large number of research results in particular from the last two decades that illustrate how successful learning support can be realized by adopting a pedagogical theory and using engineering approaches to implement appropriate computer-based tools. We can even observe families of tools (consisting of tools with similar common properties) that seem to be suitable for certain educational approaches. In the following of this introductory chapter, I describe two prominent examples for this relation between pedagogical theory and system engineering and briefly outline exemplary educational software families that have emerged. The main motivation for this thesis lies in an integration issue between these two areas: it is observable in CSCL (Computer Support for Collaborative Learning) literature that two theoretical positions shown in sections 1.1 and 1.2 are frequently combined and taken as a joint foundation or justification for educational approaches or tools. As both of these theories have some typical "established" associated software tools that are designed to support learning in the sense of the respective educational approach, the question arises in how far also on the technological side, an integration is possible and reasonable.

On a more concrete level, this question is formulated in section 1.3 and manifests in the challenge to build a flexible *Collaborative Mindtool*. This thesis deals with this integration question on an engineering level by analyzing and formulating requirements, and presenting a solution - a collaborative modeling framework - based on current technology.

1.1 Mindtools as Constructivist Learning Environments

In recent literature, we often find the claim that specific computer-based learning environments allow for constructive tasks and/or induce an active learner role. An illustration: out of 60 papers accepted for the CSCL 2003 conference, 21 explicitly mention this direction in title, abstract or introduction. 11 of these 21 present a computer-based learning environment based on that particular direction. One of the theoretical foundations frequently given for this is constructivist learning theory, which bases on philosophical and pedagogical ideas developed mainly by Piaget, Dewey, Montessori, Bruner, and Vygotsky. The common adoption of this theory is not beyond critique. Sfard (1998) generally states that all simplified models to explain learning fall short at some point and thus a restriction to one theory or

metaphor is likely to lead towards a too restricted view on complex real phenomena. Another point of criticism to the "common sense" adoption of constructivism is put by Glasersfeld (1995), who believes that the current vogue of constructivism in educational literature is to some extent superficial:

> *"Some of its advocates tout it as a panacea but would reject it if they became aware of its epistemological implications."* (page 176)

Constructivism is founded on the premise that, by reflecting on experiences, learners construct their own understanding of the world they live in (Fosnot, 1996). Learning, therefore, is the process of adjusting our mental models to accommodate new experiences. This suggests that learning is an active, rather than a passive, process.

The educational paradigm of constructivism is far from being exactly definable (in fact, a variety of definitions can be found) or even suitable for operationalization. Nevertheless, a number of attempts have been made that go towards elaborating concrete design criteria for learning environments in order to fulfil constructivist claims.

Wilson (1996, page 5) defines a constructivist learning environment as

> *" [...] a place where learners may work together and support each other as they use a variety of tools and information resources in their guided pursuit of learning goals and problem-solving activities."*

He emphasizes learning environments (as opposed to instructional environments) in order to promote a more flexible idea of learning which stresses

> *"[...] meaningful, authentic activities that help the learner to construct understandings and develop skills relevant to problem solving."* (page 3)

A number of approaches have been made to transform higher order goals into concrete guidelines that explain the design principles or goals for constructivist learning environments. Not surprisingly, these guidelines tend to be on a relatively high or general level.

Glasersfeld (1995) reflects upon constructivist implications for educational situations is general, including the critical question what teaching means when there is no such thing as objective knowledge. The four central suggestions he makes are the following:

- Teachers should help rather than instruct, the notion for educational activities should be teaching instead of training.
- The orienting function of language and perceptual material can be conductive to reflections and abstractions.
- Teachers should try to infer the thinking of their students in order to be able to orientate them and understand the conceptual changes that occur.
- Reflection, as it can cause conceptual changes, should be fostered, e.g. by applying group work phases in which the weakest student will have to present the final result.

The results of Honebein (1996) are more guiding for educational practitioners. They can be summarized briefly as follows:

- Provide students with experience with the knowledge construction process.

- Provide experience in and appreciation for multiple perspectives.
- Maintain the authentic context of the learning task.
- Allow for a student-centered learning process whereby students play an important role in setting the goals for learning.
- Provide for collaboration.
- Use multiple modes of representation.
- Encourage metacognitive and reflexive activities.

While these principles relate to the design of generic educational environments, some other approaches have put a specific focus on the design criteria for computer-based tools in order to make them usable in a constructivist sense.

A very prominent example for this viewpoint is the notion of Mindtools as introduced by Jonassen (2000). He takes up the constructivist perspective as a key foundation for the use of computer technology in education. His approach aims at fostering critical thinking by learning *with* (instead of *from*) the computer:

> *"Mindtools are computer-based tools and learning environments that have been adapted or developed to function as intellectual partners with the learner in order to engage and facilitate critical thinking and higher order learning. [...] Mindtools are constructivist knowledge construction tools."* (pages 9 & 12)

Mindtools are interactive learning environments and "computational objects to think with". They foster *meaningful learning* defined by Jonassen, Peck, and Wilson (1999) as being *active*, *constructive*, *intentional*, *authentic* and *cooperative*. The characteristics by which can be determined whether a tool is a Mindtool or not (and thus the key properties a tool should have in order to support such high-level goals as meaningful learning) as given by Jonassen are listed below. Interestingly, these criteria are associated to very different areas and in particular cover several fields of educational and computer science.

- **Computer-basedness**. While in general, Mindtools are not necessarily computer-based, Jonassen lists this requirement as his concern is computers in education.
- **Availability**. A Mindtool must be a readily available, "off the shelf" general computer application.
- **Affordability**. Mindtools should be available at low costs to allow for regular school use.
- **Knowledge construction**. Mindtools can be used to construct and represent content or personal knowledge.
- **Generalizability**. A Mindtool can be used in different domains, they are not restricted to specific areas or subjects.
- **Critical Thinking**. The use of a Mindtool engages the learner in critical thinking, which is deeper and higher order than just memorizing what someone else said about the content.
- **Transferability**. Using Mindtools results in the construction of generalizable, transferable skills that is not bound to the subject the Mindtool was used with.

Table 1.1: Mindtool examples - categories and applications

Category	Applications
Semantic Organization Tools	Databases Semantic Networks
Dynamic Modeling Tools	Spreadsheets Expert Systems Systems Modeling Microworlds
Interpretation Tools	Intentional Information Search Visualization Tools
Knowledge Construction Tools	Hypermedia Editing Tools
Conversation Tools	Synchronous Conferencing Tools Asynchronous Conferencing Tools

- **Formalism**. The formalism embedded in a Mindtool provides a simple but powerful way of thinking.
- **Learnability**. The use of a Mindtool must be easily learnable, so that the effort needed to learn how to use the tool does not exceed the benefits which result from using it.

In the sense of Jonassen, these criteria are not to be considered as absolute but as indicators of the degree of "mindtoolness" that a tool has. Some application categories with a high degree of such "mindtoolness", together with some specific example applications for each category, are listed in table 1.1 which is taken from Jonassen (2000). This table does not mean to be exhaustive. We can find several recent publications in which tools are being attributed Mindtool-like properties or are explicitly categorized as being Mindtools. Examples include the WORLDMAKER modeling tool for children (Law & Tam, 1998), the INSPIRATION concept mapping application (Dabbagh, 2001), the mathematics learning environment ELLE (Morteo & Mariscal, 2003), tools targeted towards learning the concept of time (Wang, Wang, & Huang, 2002), and some usages of Lego Mindstorms technology (Savage, Sanchez, O'Donnel, & Tangney, 2003).

1.2 Computer Support for Collaborative Learning

Compared to other "established" research areas in the educational sciences, the field of *collaborative learning* is still relatively young. The basic and simple idea behind it is that it is beneficial for learners to work together on learning tasks. It is assumed that the joint construction of meaning through interaction with others enhances learning (Littleton & Häkkinen, 1999). Evidence for this hypothesis has e.g. been given by Johnson and Johnson (1990, page 26) who state that:

> *"[...] generally achievement is higher in co-operative learning situations than in competitive or individualistic ones and that cooperative efforts result in more frequent use of higher-level reasoning strategies, more frequent process gain, and higher performance on subsequent tests taken individually than do competitive or individualistic efforts."*

As both the term collaboration and the term learning are used in a variety of meanings, a precise and universally accepted definition of the term "collaborative

learning" did not yet evolve. While the term "learning" is already used ambiguously, the meanings of the adjective "collaborative" diverge even more (Dillenbourg, 1999). For instance, some authors refer to "collaboration" whenever two or more people interact at all, while others carefully distinguish between the different nuances expressed with terms like cooperation, coordination, communication, cooperation and collaboration (Herrmann, 2001). In these cases, the difference between collaboration and cooperation is mainly seen in the fact that collaboration requires more than an effective division of labor (which would already constitute cooperation) but "real" joint activity. In this sense, Roschelle and Teasley (1995, page 70) give the following definition:

> *"Collaboration is a coordinated, synchronous activity that is the result of a continued attempt to construct and maintain a shared conception of a problem."*

Dillenbourg (1999) takes up this definition, but states that it lacks one important aspect of collaborative learning that a common theory would have to offer as a dimension: the inclusion of collaborative situations. In his analysis, he points out the four aspects of "collaborative learning" listed below.

Situations. A situation is called collaborative if it contains a certain symmetry between the actions, knowledge, and status of the participants. Furthermore, collaborative situations are often characterized by shared goals among the participants and a low division of labor (i.e., participants solve a task together instead of dividing it into independent subtasks, merely assembling partial solutions).

Interactions. Collaborative interactions are characterized as being interactive (individual actions are interwoven), synchronous (though a sharp distinction between what is still synchronous and what is already asynchronous is hard to make) and negotiable. The latter means that in collaborative interactions, participants are expected to justify their positions and argue for their standpoints.

Processes. Learning mechanisms in collaborative learning definitely involve those of individual cognition (e.g. induction, cognitive load, or conflict), one might hope that collaboration is a key for making these processes happen more often. However, there are also processes that are specific to collaborative situations. Examples given by Dillenbourg include internalization, appropriation, and mutual modeling.

Effects. Methodological issues around the question of measuring effects of collaborative learning deal with, e.g., group performance measurements versus individual measurements. Also the choice of dependent and independent variables, influenced by the availability of detailed interaction data (through recordings done by CSCL systems) is of interest here.

As opposed to other learning concepts (including the constructivist approaches as presented in the previous section), research in the area of collaborative learning has already established quite close connections to more technological oriented fields. Koschmann (2002, page 20) has given a definition for the research area of computer supported collaborative learning:

> *"CSCL is a field of study centrally concerned with meaning and the practices of meaning-making in the context of joint activity and the ways in which these practices are mediated through designed artifacts."*

Not surprisingly, a lot of these "designed artifacts" employ modern networked computer technology to support one or more aspects of collaborative learning. The role of the computer in these collaborative learning settings varies considerably. Extreme examples are the pure usage of computers in traditional face-to-face learning scenarios as well as approaches in which the computer is intended to be a collaboration partner (Dillenbourg & Self, 1995). The following roles can be found more frequently.

Direct collaboration means. In a lot of CSCL systems, the most important purpose of the computer is to enable or support the collaboration between the learners on a technical level. Typical examples range from pure communication support (e.g. through chats or discussion forums) to co-constructive environments. A very early example of the latter type are the COGNOTER and ARGNOTER tools developed in the Colab at Xerox PARC (Stefik et al., 1987).

Indirect collaboration mediator. In collaborative learning, the computer can also be used to build and maintain models of the learners. These user models, together with a group model, can be of value to support collaborative processes in the group, e.g. by recommending suitable peer learners (Hoppe, 1995), by dynamically building groups that are likely to benefit from collaboration (Ikeda, Go, & Mizoguchi, 1997), or through feedback processes which can allow for community building (Suzuki & Funaoi, 2002). In all these scenarios, the computer has a mediating role - though one could reasonably argue that more sophisticated systems in this category are also means of intelligent support (see below).

Interaction analysis. If the computer is used as a collaboration means, detailed log files about the interaction that took place are usually available. Using these files (or even "live" data from ongoing collaboration processes) as input for analysis methods as done, e.g., by Mühlenbrock (2001) and Soller and Lesgold (2003) enables insight into the interactions that occurred, and may therefore be of value, e.g. to understand the group dynamics with respect to the collaborative task.

Intelligent Support. The results of interaction analysis can be used to provide the learners with feedback on their collaborative task. Jermann, Soller, and Mühlenbrock (2001) have given a good overview on existing applications that apply such "guiding" mechanisms. Considering the enormous complexity and variety of human interactions, the intelligent support for collaborative learning is still in its infancy. In fact, it is reasonable to argue that also within the roles of the computer as direct or indirect collaboration facilitator, a number of intelligent collaboration techniques can be embedded without an explicit system intervention.

For further, more systematic, investigations about the possible advantages of computer technology in collaborative learning, fundamental and general research results about factors for effective collaboration are necessary. For the *process-oriented* dimension of collaborative learning, Linden, Erkens, Schmidt, and Renshaw (2000) suggest the following factors.

- *Maintaining common ground* - participants must be aware of the task goal and stay in common focus
- *Co-responsibility, equality and mutuality* - participants must have a significant role and be co-responsible for the overall task.

- *Mutual support and criticism* - a central point for the effectiveness of collaborative learning, which makes learners reach higher goals than they could individually.
- *Verbalization and co-construction* - the externalization of knowledge helps students with the performance of cognitive processes.
- *Elaboration* - students can learn themselves by acting as peer helpers.
- *Tuning in cognitively and socially* - as learners are more at one level of understanding than teacher and learner, it is assumed that communication within a learner group can be more effective than teaching.

As Veerman and Treasure-Jones (1999) point out, supporting collaboration with computer-based tools can be successful if it aims at supporting one or more of these factors. They explicitly mention the usefulness of general communication tools in the sense that these already support the criterion of *verbalization* by allowing the learners to negotiate on the common task.

Joolingen (2000) takes up this idea and states that some factors of collaborative learning can be supported more or less independent of a concrete task by features of existing general tools. One of these general features is e.g., the sharing of resources. If an environment enables an exchange and joint creation of material, this is likely to contribute to the criterion of *co-responsibility*.

Applications that rely on shared workspaces and enable the users, usually by means of direct manipulation, to "communicate through the artefact", are likely to contribute to the criterion of *co-construction*, as Dix, Finlay, Abowd, and Beale (2004, page 690) state:

> *"The lesson [...] is that cooperation does not necessarily involve direct communication and, even where it does, the indirect channel through the artifact may be central to effective working."*

As argued, at least the criteria of verbalization, co-responsibility and co-construction can be supported by general, task-independent tools. Specifically designed applications can of course reach even higher levels. One perspective on this thesis is that it illustrates the implementation of a relatively flexible framework that effectively contributes to reaching some of these higher levels

1.3 The Collaborative Mindtool Approach

As presented in the two previous sections, the approaches of constructivist learning environments and collaborative learning are not unrelated, neither from theoretical foundations nor from practical positions. A lot of CSCL environments base on constructivist learning approaches, and the aim of allowing the learners to co-construct meaning is frequently formulated. In turn, we find statements that the provision for collaboration is among the design criteria for constructivist learning environments and that communication technologies can realize constructivist ideals of learning (Bonk & Cunningham, 1998). Also the example Mindtools list contains one entry (conversation tools) in which a single-user mode does not make sense.

However, the majority of research results that deal with the relation between constructivist learning and collaborative learning are on a conceptual or theoretical level. Although for both areas, potential benefits gained by computer support are known, technological implementations with a clearly integrative aim are rare.

There are, indeed, some applications that are designed to meet both collaborative and constructivist approaches. Examples include the following:

- In the area of language learning, Weasenforth, Biesenbach-Lucas, and Meloni (2002) have explored the use of threaded discussions as a means to reach constructivist learning goals.
- Savage et al. (2003) have used Lego Mindstorms technology as a Mindtool in a collaborative setting (Lego Mindstorms, n.d.).
- The CoVASE application (Jensen, Seipel, Nejdl, & Olbrich, 2003) for collaborative visualizations of complex virtual experiments.
- Conducting studies about constructivist collaborative learning, Hübscher-Younger and Narayanan (2003) have used the CAROUSEL tool for collaborative algorithm representation.
- A number of collaborative concept mapping tools that have been used in varying contexts with diverse aims (Cicognani, 2000; Kuan, Lee, & Ho, 2003; Silander, Sutinen, & Tarhio, 2004).

As this list may suggest, the (few) present tools and environments are either domain specific, or asymmetric in the sense that they explicitly focus on one of the two directions. The absence of a variety of general and systematic integration approaches surprises, as especially the Mindtool notion given by Jonassen (2000) seems to be designed for further investigations from a CSCL perspective.

Hoppe (2001) introduced the notion of *Collaborative Mindtools*: tools that inherently synthesize communication and collaboration support with interactive and constructive features. In (2004), he illustrates three practical examples (including two which technically base on earlier versions of the software described within this thesis) and outlines general benefits gained by combining techniques of computer based communication with interactive cognitive tools. His characterization of Collaborative Mindtools is:

> *"In summary, the notion 'collaborative mind tools' stands for a new tendency to extend CSCL technology beyond language centred computer-mediated communication towards richer environments providing 'computational objects to think with' - now jointly."* (page 226)

In a related area, Joolingen (2000) has also proposed such an integration: introducing his approach of *collaborative discovery learning*, he analyzes potential and criteria for computer support that takes into account both discovery and collaborative learning principles. He expects that

> *"[...] the discovery behavior displayed by learners may improve under influence of collaboration. On the other hand, collaboration, and especially the communication that underlies it, may benefit from information that can be extracted from the discovery process."* (page 204)

Driving this approach forward, Joolingen presents a high-level software architecture that embeds both experimentation spaces and collaboration tools.

Another proposition that deals with integration of cognitive tools in collaborative environments has been given by Milrad, Spector, and Davidsen (2002). Their approach of *Model Facilitated Learning* contains a conceptual framework for the integration of modeling and simulations into rich learning environments. Emphasizing the diverse options of using models and modeling in learning situations, they explicitly mention the need for supporting collaboration, interaction and reflection "around and beyond" simulations.

Milrad et al. explicitly refer to the modeling means of causal loop diagrams (Senge, 1990) and System Dynamics (Forrester, 1968). These techniques clearly

fulfil the Mindtool criteria of Jonassen - in fact, dynamic modeling tools even constitute a separate category in his classification of examples. Joolingen illustrates his approach with the integration of a domain knowledge component into the BELVEDERE argumentation tool (Suthers, Toth, & Weiner, 1997). The resulting environment also fulfils the core Mindtool criteria, leaving out some practical issues like availability. Abstracting from the concrete BELVEDERE example, we can argue that the experimentation spaces have a high "mindtoolness" potential: they are designed to support discovery learning and encapsulate a domain model. Thus, they are intended to engage the learner in interacting with the underlying semantics and formalism of the model, trying to make meaning - which is not far from the essence of Mindtools.

As shown, a common element between the two approaches of collaborative discovery learning and Model Facilitated Learning is that they, without explicitly mentioning it, demand for research and development in the area of Collaborative Mindtools - computer-based tools that fulfil the Mindtool criteria and provide a generic collaboration support.

The construction of a system that allows for this kind of integration is the core motivation for this thesis from a pedagogical point of view. From an engineering point of view, this challenge does not only translate to the enhancement of some existing Mindtool with specific collaboration support features. Moreover, taking into account established research and development principles of computer science and in particular software design, the challenges lie in a combination of the following four requirements:

1. The system should contain a generic support for collaborative activities and, in particular, for co-constructive processes.

2. The generic design of the collaborative system should not be sacrificed for specific integration needs (Roschelle, DiGiano, & Chung, 2000) - on the other extreme, the system should still be simple and expressive enough to be really useful for practical issues (and not "just" a high-level architecture).

3. The software system to be developed should be *easily extensible* and can, through these extensions, be used as a Mindtool in flexible ways, potentially even covering several of the categories that Jonassen lists.

4. An integrated and flexible use of these different system extensions should be possible (Roschelle et al., 1999). This does not intend to claim that integrating different tools is always beneficial for the user - however, the system should allow for the construction and use of such integrated tools.

Considering in particular the fourth criterion, the trade-off between generality and expressiveness of the intended system is one of the critical design questions. It is obvious that the targeted application range that the framework is designed to support does have an impact on the answer to this question. A too wide range (i.e. the claim to integrate all possible Mindtools with good and generic collaboration support) risks losing expressiveness. On the other hand, an area too small (which could result from restricting the focus to one specific tool like, e.g., spreadsheets) may be criticized for lacking flexibility. The next section describes and motivates the choices made.

1.4 Collaborative Modeling with Graph Based Representations

In science, the term*model* refers to a schematic, simplified and idealized representation of an object or a domain, in which the relations and functions of the elements of the objects are made explicit. There is an analogy between the model and the object it describes in the sense that these two are structurally identical (Meyers Enzyklopädisches Lexikon, 1976). Modeling is understood as the activity of creating, manipulating and using models.

As models are a simplified and manageable means of understanding complex real phenomena, the importance of modeling in science education is evident. Bredeweg and Forbus (2003) support this position and additionally emphasize the function of modeling as a means of knowledge externalization:

> *"Modeling is a central skill in scientific reasoning and provides a way of articulating knowledge. Learning to formulate, test, and revise models is a crucial aspect of understanding science and is critical to helping students become active, lifelong learners."* (page 35)

Also in the specific area of learning environments that emphasize a constructive learner role, the potential for modeling as a powerful learning technique is recognized (Jonassen, 2000; Perkins, 1991).

A general function that computers can have in the domain of modeling is that they can serve as tools that execute models or run simulations that are based on models. Both is possible for many formal modeling languages like e.g. Petri Nets (Petri, 1962) or System Dynamics (Forrester, 1968). Interestingly, also some qualitative models can be "run", as Biswas, Schwartz, and Bransford (2001) show with their implementation of generating explanations and "teachable agents" from concept maps.

In a general sense, the idea of using computers as active tools for modeling and running models is indeed not new: Kay and Goldberg (1977/2001) described their vision of a Dynabook - a notebook size "self-contained knowledge manipulator" that can, among functions for displaying different media types, serve as an integrated repository for dynamic media and simulations. The hypothetical device that Kay and Goldberg describe offers the option to run these simulations and to interact with them. Their vision includes educational usages:

> *"Mathematics could become a living language in which children could cause exciting things to happen. Laboratory experiments and simulations too expensive or difficult to prepare could easily be demonstrated."* (page 177)

With technology as available today, these usages of computers for modeling and simulation are possible and established - with some right, these usages could be called "basic support for modeling". More sophisticated functions of computers within modeling tasks have been worked out. It has also been shown that from an *educational* perspective, there are specific reasons for supporting modeling activities with modern computer based tools (Bredeweg & Forbus, 2003; Milrad et al., 2002; Jackson, Stratford, Krajcik, & Soloway, 1996). Specific reasons include the following:

- The Microworlds idea of Papert (1980) in which computer-based tools (he does not directly call these modeling tools, but this classification would fit well in most cases) would provide children with a whole range of transformative developmental experiences. He imagined that constructions within

these powerful computing engines would enhance children's imaginative and intellectual power.

- Wild (1996) has argued that, as computer modeling externalizes thinking and knowledge, models become tools for the conscious manipulation of thought.
- Dynamic models which can be "run" are of more value when used with a computer. Kurtz dos Santos and Ogborn (1994) argue that the increase of flexibility and interactivity that computers offer (e.g., certain parameters can be changed quickly and the result can be compared to the original state) enhances learner's understanding of problem.
- Sometimes, a modeling task can benefit from the use of multiple representations. The exploration of these different representations can be beneficial for learners (Ainsworth, 1999), and computers enable or facilitate the transitions between different representations greatly.

Digital technology can also, through archival and retrieval functions, foster the exchange and re-use of modeling material. Networking also principally enables the cooperative use of modeling tools.

The question of the specific kinds of appropriate or needed support that computers can offer for collaborative modeling tasks is not completely answered and an issue of current research. Sierhuis and Selvin (1996) describe some general criteria they consider as necessary for successful modeling support in a collaborative project. For the task of *team based system perception and construction of static models*, they mention the following ones:

Create meaning. The modeling technique used should enable the creation of external representations that reflect meaning created by the participants.

Shared understanding. The joint creation of the external conceptualization is likely to lead to a shared understanding among the group members. Scaffolding this process is an important issue.

Create structure. A framework to guide the shared modeling process is needed to assert the group members a certain process structure.

Communication. A shared conceptualization will allow the users to talk about the domain without ambiguity and confusion - yet this communication must be enabled technically.

Reduce complexity. The modeling method used should be suitable to reduce the complexity of a certain domain or situation.

More recent contributions to the area of collaborative modeling, which also emphasize the learning perspective, have been made by Joolingen and Löhner (2001) and Or-Bach (2003). The former paper focuses on the use of different representations (textual, graphical or output oriented) for collaborative modeling tasks, the latter reflects about design decisions in terms of interaction modes and support mechanisms for computer based modeling tools.

Treating the domain of computer technology for modeling, it is not a surprise that both papers address the function of the computer as an active medium that can be used to run simulations generated from the constructed models. This level of support can be classified as *task support*. Indeed, a second joint position of the papers is that they see the basic means for supporting collaborative modeling in *process support* (cf. 1.2), namely in enabling the learners to jointly work on the model and co-construct it. This technique of sharing of external representations

with the option of communicating through the constructed artefact has been mentioned before in this thesis as an approach that fulfils both constructivist criteria and positions from collaborative learning. An implementation of this idea with suitable external conceptual representations (i.e. modeling languages and techniques), eventually enhanced with additional communication features, it likely to fulfil the methodology criteria of Sierhuis and Selvin (1996).

External representations and their functions are a research field within cognitive science. Within a recently published analysis of the use of external representations for modeling, Löhner, Joolingen, and Savelsbergh (2003) have presented a review on literature in this field and summarize the different general functions of representations as follows:

- They serve as an extension of the internal working memory.
- Different representations make certain aspects more or less accessible and visible and thus give a layout for a problem space.
- External representations can anchor and determine cognitive behavior by allowing or restricting certain cognitive actions.
- In situations that involve multiple users, representations have a communication function by helping to express ideas.

In collaborative contexts, shared representations have some additional functions. Hoppe and R-Plötzner (1999) have identified the following effects of shared representations on shared cognition:

Coordination. The coordination of individual contributions is both constrained and mediated by the external environment.

Reification. Contributions are given objective, material evidence in the external environment.

Illustration. In addition to their content, the individual contributions may be illustrated by the external representation.

Storage. The external environment may store individual contributions for later use, e.g., for the purpose of reflection.

"Mise en relation". The external environment relates different contributions to each other in an objective way.

Suthers and Hundhausen (2003) confirm these functions of shared representations. They add the following two:

- The manipulation of external representations can initiate negotiations of meaning: the negotiation can be prompted by potential for action offered by the representation before that action even begins.
- By means of gestural deixis, the individuals can use the developed external representations as a proxy for the ideas that they represent.

It is a known fact that even in the individual case, not all representational notations fulfil each of these functions in the same way (Larkin & Simon, 1987; Zhang, 1997) and thus the used representation may have an influence on a task result. Suthers (1999a, 1999b) has motivated theoretically and empirically that for the case of *collaborative representations* (i.e. collaboratively created and manipulated external representations), even more specific effects occur: the representation used

for the collaborative activity biases the collaboration process itself. In a prominent example (Suthers & Hundhausen, 2003), he compares groups engaged in a task of collaborative scientific inquiry. He shows that the quality of the group results differs considerably depending on the representation the group used for argumentation (text, matrix of graph).

Löhner et al. (2003) conducted some similar studies in the domain of modeling complex phenomena: students had to solve a dynamic modeling task collaboratively. One group of them used a textual representation and typed in the formulae, the other group used a graph representation similar to System Dynamics. Additionally, both groups had the option to run simulations based on their model. The study results presented by Löhner et al. (2003) address the general affordances of the representations for the collaborative modeling task (the specific impacts on the character of the collaboration process are not deeply analyzed). These results seem to confirm the findings of Suthers: the modeling representations strongly influenced the result of the collaborative task.

The specific results of the study are manifold and generally seem to encourage the use of visual, graph based representations for modeling - at least for dynamic modeling techniques as used by Löhner et al.. Their detail analysis includes the following:

- The quality and complexity of the constructed models was much higher in the case of the graph based representation.
- The graph based representation was more inviting to use as an external working memory: it seemed to be easier to use and express knowledge in. On the other hand, for experienced users it sometimes seemed too restricting.
- The graph based representation invited more for trying out different solutions. Though this was not always directly guiding the learners towards the correct solution and is thus a negative argument from a formal modeling point of view, it can definitely be seen as a general advantage from a constructivist educational perspective.
- Although the learners with the textual representations were able to reason deeply and systematically about the task, they had serious difficulties in putting their ideas into practice.
- There is an indication that learners would benefit from mixed representations, allowing both easy experimenting and expression power.
- An ideal representational system should provide for smooth transitions between qualitative and quantitative phases.

As motivated in the previous section, one aim of this thesis is to describe the implementation of a flexible and expressive framework to support heterogeneous Collaborative Mindtools. It was also argued that a certain restriction of scope might be needed to retain expressive power. The critical design question is whether a restriction to graph based representations is reasonable or not. Some imaginable reasons against graphs as representational primitives are that they are less compact than texts (requiring more space), and require the user to make explicit relationships which can be implicitly expressed in text structures. Furthermore, simply typing in a formal semantics might be easier than building it with a graph. Indeed, some detail findings of Löhner et al. challenge a positive answer to the question whether graphs as representational primitives are really suitable. The use of graph based representations, as opposed to textual representations, sometimes seemed to restrict the insight into the created model and was partially hindering users in expressing

their knowledge. However, the result of their study as a whole seems to suggest a positive answer. In other sources, in particular also from different areas of research, we can find additional confirming evidence:

- The results of Suthers and Hundhausen (2003) generally motivate that visually structured representations seem to provide guidance for collaborative learning. His findings specifically include that graph based representations seem to invite learners to elaborate on created structures, and that they seem to be suitable for remembering.

- The variety of modeling languages that relies on graph based representations (or that can, as one alternative, be represented in such a notation) is impressing. More formal languages with exactly defined semantics are, e.g., Petri Nets (Petri, 1962), System Dynamics (Forrester, 1968), entity relationship diagrams (Chen, 1976), finite state automata (Hopcroft, Motwani, & Ullman, 2000), or Bayesian networks (DeGroot, 1989). Languages with an intermediary level of formality (i.e., some parts of the expressions allow for an automatic interpretation and eventually simulation, while others do not) include most diagram types of the unified modeling language UML (Booch, Jacobson, & Rumbaugh, 1998), causal feedback diagrams (Senge, 1990), or hypermedia editing languages like in XCHIPS (Wang, Haake, Rubart, & Tietze, 2000). Finally, there are also a lot of qualitative modeling techniques that make use of graph based representations. In those, the object and link *types* can usually be exactly distinguished and are possible subject of interpretation, the *content* of these objects and links is however usually not accessible to computer based interpretation techniques. Examples of this category include mapping techniques like concept mapping (Novak & Gowin, 1984; Dabbagh, 2001) or mind mapping (Buzan, 2002), and the design rationale method QOC (MacLean, Young, Bellotti, & Moran, 1991).

- Also apart from modeling in the narrower sense that a formally defined modeling language is employed, graph based representations are widely used, in particular also collaboratively and/or in areas that go well with the Mindtool approach. Here, prominent application areas include discussions (Gaßner, 2003), and argumentations (Suthers et al., 2001; Stefik et al., 1987). Collaborative modeling approaches with graph based techniques that are self defined, deeply domain related and unstandardized but indeed helpful to learners have recently been demonstrated in the areas of stochastics (Lingnau et al., 2003), seismology (Baloian, Breuer, Hoppe, & Pino, 2004), and astronomy (Hoeksema, Jansen, & Hoppe, 2004). The latter three scenarios have used the framework described within this thesis.

- Any modeling technique that is accessible with a graph based representation induces a certain structure on the models created with that technique: they consist of objects and links of some *type*. The specific types are often fixed and predefined, and sometimes supplemented by syntax constraints, to allow a certain system-side insight into the created representation. With two languages that both employ dedicated object and link types and thus offer *structure*, connections and transformation between these languages can be enabled on very fine granular level. In fact, this structure that is available for graph based modeling representations but *not* within general modeling tools, is what enables tight and at the same time flexible integration. It allows, e.g., a system dynamic network that simulates traffic jams, to receive input from several Petri Nets that model the logic of traffic lights - without having to define a general transformation between these two different languages, some partial mapping rules are sufficient.

- Also due to the structure that is inherent to graph based representations, flexible ways of cooperating by using shared models arise. Examples include:
  - the sharing of single objects or whole subgraphs, either defined through some neighborhood criteria, or even by semantic type - the latter allowing e.g. generic support for private comments attached to a jointly used model.
  - "jigsaw" designs (Aronson, 1978) in which the different available object types are distributed among the participants (e.g., one learner can add classes to a UML diagram, another one interfaces, and a third the relationships). As no learner can construct a complete solution on his own, this method is expected to induce and enhance collaboration.
- Aiming at supporting collaborative modeling in general is likely to result in relatively high level architectures or protocols, because little is known about concrete tools or representations. A restriction to graph based structures meets requirements coming from modern object oriented software design quite well and allows a more detailed computational abstraction. In addition, established principles from graph theory (Berge, 1976) can be applied already on the framework level, which instantly brings advantages for all supported tools.

This section aimed at showing two things. Firstly, it has been argued that modeling is a good candidate for a collaborative Mindtool. Secondly, I tried to convey that graph based representations for modeling are commonly used and often reasonable. From a computational point of view, some reasons were given that the loss of generality that inevitably occurs when restricting a framework for collaborative modeling tools might be compensated by gained structure, which can constitute a base for interoperability, integration and generic collaboration support.

1.5 Challenges and Aims

In this final section of the introduction to this thesis, I want to briefly summarize the aims of this thesis as developed before, and the challenges that have to be faced in order to reach these aims.

As pointed out in section 1.3, the motivation for the work in this thesis is to build a flexible collaborative mindtool that supports *co-constructive modeling activities*. The choice of modeling as a target activity is supported by various authors, such as Bredeweg and Forbus (2003), Jonassen (2000), Perkins (1991), Papert (1980), or Wild (1996) who state the importance of modeling in educational contexts. The value of collaboration in modeling contexts has been pointed out by Sierhuis and Selvin (1996), and specific contributions to the educational dimensions of collaborative modeling have been given by, e.g., Joolingen and Löhner (2001), and Or-Bach (2003).

A number of authors point out that in the context of collaborative modeling (in particular in education), two design principles are key factors for success:

- *Flexibility* with respect to the supported representations, to support multiple representations of problems, different phases in the activity (which correspond to different representations), and dynamic interactive changes of simulation parameters (Löhner et al., 2003; Ainsworth, 1999; Kurtz dos Santos & Ogborn, 1994).

- *Interoperability* between different models and different model representations, to enable mixed structures and flexible creation of customized expressions with "building blocks" (Löhner et al., 2003; Roschelle et al., 1999).

In particular concerning the second point, several interoperability requirements can be distinguished:

Syntactic interoperability. The framework should allow the use of mixed external representations and must therefore be able to deal with the heterogeneous elements represented within the graph structures that occur in these cases. A basic requirement is that it should be possible to connect arbitrary elements - and also to restrict or control these options in a suitable way if desired.

Semantic interoperability. Going beyond the ability to allow for *mixed* representations, the framework should foresee tool *integration* issues. The design goal is to allow tools to reuse elements defined within others, share semantic definitions and also offer support for easy *transformations* of expressions from one language to another (Heiler, 1995; Read, Verdejo, & Barros, 2003).

Social interoperability. Given the targeted educational use, the framework must be easy to operate, and also foresee an easy use of the embedded tools. It should allow for an integrated collaborative use of different tools in a way that enables the co-learners to remain in their social work context, even when switching to other means of expression. In the context of applying mobile devices in educational settings, Milrad, Hoppe, Gottdenker, and Jansen (2004) have named this feature *educational interoperability.*

Task interoperability. Details of task support and interoperability can clearly not be treated on a framework level, as they are dependent on specific tools or even tasks. The dimensions of task interoperability that a framework can support are the avoidance of tool breaks (in analogy to media breaks), a flexible collaboration support to allow for task-compliant collaborative settings, and the support for work phases that go in line with the needs of different modeling tasks. To implement these ideas of collaborative task support (and also those of social interoperability), the framework should provide generic collaboration process support (both within itself and induced into the supported tools) by suitably enabling the learners to co-construct models. Also, there should be an explicit yet flexible support for phases in the collaborative modeling process. Apart from this generic support, collaboration can also be supported by means of specific communication oriented graph based representations, e.g. for argumentation or discourse, that can be included in the framework in the same way as the modeling tools. For further collaboration support (e.g. interaction analysis), the framework should contain suitable interfaces to connect external components.

Apart from these basic interoperability requirements, some very important design parameters for the practical implementations within this thesis are dependent on the usage scenario: considerable parts of the system (though not necessarily its computer science foundations) are influenced by the targeted learning scenarios. Here, two typical views are possible:

- The tool *is* a learning environment.
- The tool is *part of* a learning environment.

Figure 1.1: A collaborative modeling tool supporting an educational face-to-face scenario

The first position obviously has its justification, e.g. in fields like distance education. Here, the learner and the computer system are the only "participants", and (in the case of collaborative environments) the tool is also the only means of communication between the participants. In this philosophy, the success of the tool can be defined directly dependent on the learning effect on the user's side, as other factors do not play a significant role.

The second position, where a computer tool is conceived as being part of a learning scenario, but not the only determining factor, is adopted by a number of educational practitioners. On a theoretical level, justifications for this premise have, e.g., been given by Hoppe (2005, to appear) and Goldman (1996), who emphasizes the importance of social glue in science learning:

> *"The complement of Dynagrams studies documents that students can become engaged in more science practices and conversations when they have access to each other, well-planned challenges, and resource-rich materials such as manipulatives and the computer modeling environment."* (page 74)

In this pedagogical approach, the computer tool has an orchestrating role, is used occasionally and where appropriate, and co-exists with other tools - also physical ones. An even more important point is that educational decisions made outside the tool (e.g., by a teacher), social factors (e.g. group constellations), and the physical environment play a central role. Figure 1.1, taken from Hoppe (2002), illustrates nicely this approach: students, here shown in a mathematics lesson, can interact with each other in a variety of ways (directly or via the computer based tool), the

machines are embedded in the learning environment, and the software systems are used together with other traditional ones (like paper and pencil).

It is obvious that in these settings, an evaluation of the "value" of the software tool is not easy, because the usage scenario plays an important role and includes a whole range of variables. Yet, as the underlying principle of "priority of pedagogy over technology" (Hoppe, 2002) is indeed highly relevant for current educational practice, I have decided to adopt it within this thesis - taking into account that this makes a systematic general evaluation of the isolated tool (concerning the question: "does it support learning") significantly more difficult. This problem is even increased by the fact that my target system is a framework designed to support a multitude of modeling languages, and these even in an integrated manner - thus, a suitable evaluation must be independent of the employed languages.

Having named the general motivation and pedagogical target scenario, I want to conclude this summary section with a list of concrete challenges that lie in the development of a methodology and an implementation for an interoperable collaborative modeling system of the described type. Here, different *types* of requirements can be distinguished according to the following general challenges:

- Interoperability of modeling languages in a general sense (Dolk & Kottemann, 1993).
- The computational design of interoperable tools in educational contexts (Roschelle et al., 2000).
- Educational software components and their specific needs (Roschelle et al., 1999).

Concerning interoperability of modeling languages, a number of requirements have been identified by Dolk and Kottemann (1993). Rooted in database theory and structured modeling (cf. subsection 2.3.3) but exceeding the scope of these approaches, they discuss various approaches of model integration (organizational, definitional, procedural, and implementation-related). Within this framework, they develop a list of criteria that they consider necessary for an integrated modeling environment:

1. A uniform internal model definition scheme that is capable of representing many classes of models.
2. Conversion mechanisms that allow external model definitions to be transformed into the internal scheme.
3. Robust typing and inheritance at both the variable and the model level.
4. A model manipulation language that is based on message passing and allows the integration of external libraries.
5. Model solution libraries and transformation routines that allow the conversion of internal data structures to the external algorithm libraries.
6. A graphical user interface and appropriate views for supporting model definition and integration.

Apart from this requirements list, Dolk and Kottemann (1993) also pose some questions that they see as open, even when restricted to the technique of structured modeling. In particular, they mention the question of a suitable (complete) set of

primitives for a model manipulation language, and the open issue whether there is a set of necessary and sufficient abstract data types and inheritance hierarchies for modeling.

While Dolk and Kottemann mainly present thoughts about model integration in general, a number of aspects closely related to the *computational design* of interoperable tools in *educational usage contexts* are listed by Roschelle et al. (2000), who report on a project in the field of mathematics education. Their findings include the following:

1. The value of external tools (which they call "found") should not be estimated too high. Instead, they suggest a focus on few but powerful and generative tools, embedded in high-quality libraries of well-designed tools.
2. Programming conventions should be made explicit, and design patterns should be used in order to attract developers to use the system. Roschelle et al. (2000) state that these code-level issues can enormously increase re-use, as they encourage programmers.
3. On the detail level, a number of basic functions and mechanisms must simply be available in order to allow for interoperability. In particular, Roschelle et al. (2000) name the following:
   - Dynamic publishing and subscribing mechanisms to allow for interoperability at runtime.
   - Change coordination patterns to propagate events.
   - Advanced component persistence mechanisms (e.g., XML based) that exceed the capabilities of class serialization.

There are a number of common points between Dolk and Kottemann (1993) and Roschelle et al. (2000), including in particular the requirement of protocols for change notification. A major difference, however, is that Roschelle et al. deemphasize the claim of being able to integrate arbitrary externally found components, and instead focus on smaller but better designed libraries.

All the points listed above are related to more or less technical aspects, either on the design level or on the implementation level. The requirements brought up by Roschelle et al. (1999) are derived from more *educational dimensions*. Their vision is a modular system that is able to connect "educational software components". The latter are characterized as

> *"[...] high-level computational objects that are available as tangible building blocks."* (page 51)

According to Roschelle et al. (1999), the challenges in building these educational software components include the following:

1. Standards (technical, educational, curricular, and conceptual) should be used wherever possible in order to allow for plug-and-play operations among components.
2. The incorporation of new components into a system should be as simple as copy and paste operations.
3. An explicit focus should be set on enabling developers to re-use functionality of other components in the system, to reduce redundant developments.
4. There should by advanced and dynamic wiring options between components, and these wiring options must be made clear to both the programmer and the user.
5. Interoperability should not be restricted to wiring options, but must also cover domain concepts.
6. Translators and wrappers should be used to adapt existing external resources.

As a final summary of this introduction, the scope and target for this thesis can be sharpened as follows: the overall aim is to define a methodology that allows for collaborative modeling with heterogeneous and interoperable graph based structures, and to implement a flexible collaborative framework that implements this methodology. There should be an easy definition mechanism for specific representational languages, and limitations on the expressiveness of the supported graph based representations must be avoided wherever possible. This condition is due to the fact that implementations of formal modeling languages with clear operational semantics may need to apply arbitrary algorithms in order to calculate expression semantics. Apart from general issues of software engineering, specific challenges in building the methodology and the system lie in the desired levels of interoperability and flexibility - more specifically, critical design choices can be expected with respect to data structures for interoperable heterogeneous models, dynamic mechanisms to wire components, interfaces to external components, and easy to use plug-in mechanisms.

Chapter 2

Theoretical Foundations

Having presented the general motivation and scope of this thesis in the previous chapter, the following parts of this thesis are intended to give an overview about theory and existing technology in fields close to collaborative modeling with graph based representations. The aim of this chapter is to outline the theoretical background in three relevant areas: graphs, visual languages, and meta modeling approaches.

The contributions of these single areas to this thesis can be characterized as follows: *graph theory*, as a subdiscipline of mathematics, offers a conceptual view on abstract graph structures including solid terminology definitions and algorithmic approaches. This is supplemented by the research field of *visual language theory*, which explicitly considers visual representations and offers descriptive and process-oriented formalizations for these. The aspects of *meta modeling* presented in this chapter relate to descriptive, transformative and integrative perspectives on models, focusing on graph based models.

Together, graph theory and visual language theory serve as foundations for visual typed graphs, which play a key role in the approach for collaborative modeling to be presented in chapter 4. The basic contributions of meta modeling are twofold: on the one hand, the corresponding theory serves as a background for the conception of "Reference Frames" as abstractions of modeling languages, and on the other hand the integration and transformation oriented approaches are a considerable input concerning interoperability issues in heterogeneous models.

2.1 Graph Theory

Large parts of the implementations within this thesis essentially make use of graph based representations and their use for modeling. For that reason, a systematic investigation of the theoretical foundations in these fields is necessary. The aims of these investigations are threefold:

- to understand the concept of a graph, as used in relevant fields of research,
- to review existing formalisms and techniques operating on graphs that are worth considering for the implementations within this thesis,
- to obtain criteria for a systematic comparison of existing technology, which will be done within the chapters 5 and 3.

Since this thesis targets modeling with graph based representations, the theory of graphs as such is worth considering. This theory belongs to the field of discrete mathematics and has a quite unusual development, from a collection of scattered

and seemingly unconnected problems and puzzles (like, e.g., the Königsberg bridge problem) in various areas to a unified theory that started its rapid development in the 1950s (Berge, 1976). Today, largely due to the usefulness of graphs as models for computation and optimization, graph theory is a field of intensive mathematical studies that gains considerable interest (Gross & Yellen, 1999). All definitions within the following subsections have been taken from Berge (1976), Gross and Yellen (1999), and Aigner (1993), though any other introductory book on graph theory would give equivalent definitions.

2.1.1 Basic Definitions

As a discipline of mathematics, it is not surprising that graph theory offers a set of definitions that provide a formal and unambiguous fundament for further investigations. The base definition for graph theory is of course that of a *graph*, which is a structure that consists of elements and links between these elements.

Definition 2.1 *A* graph *G is defined to be a pair (N,E), where*

1. $N = \{n_1, n_2, ..., n_i\}$ *is a finite set of elements called* vertices*, and*
2. $E = \{e_1, e_2, ..., e_j\}$ *is a finite family of pairs (x,y)* $\in N \times N$*, called* edges*. An element (x,y) can appear more than once in E. A graph in which no element of* $N \times N$ *appears more than p times in E is called a* p-graph*.*

As expressed in this definition, there is no restriction concerning the multiplicity of edges in a graph. For some applications, it is reasonable not to allow multiple edges and loops (edges from a vertex to itself). This is done in simple graphs.

Definition 2.2 *A 1-graph G=(N,E) with*

$$\forall n \in N : (n, n) \notin E$$

is called a simple graph*.*

Example 2.1 *Let* $N = \{a, b, c\}$*, and* $E = \{e_1, e_2, e_3, e_4\}$ *with* $e_1 = (a, b), e_2 = (b, a), e_3 = (c, c)$*, and* $e_4 = (a, c)$*. Then G=(N,E) is a graph with three vertices and four edges. G is not simple because* e_3 *is a loop.*

Graphs as introduced in definition 2.1 are directed because they distinguish between start and end vertices of edges. Undirected graphs are structures that do not make this distinction:

Definition 2.3 *An* undirected graph *G is a pair (N,E), where*

1. $N = \{n_1, n_2, ..., n_i\}$ *is a finite set of elements called* vertices*, and*
2. $E = \{e_1, e_2, ..., e_j\}$ *is a finite family of 2-element multisets* $\{x, y\} \subseteq N$*, called* edges*. An element {x,y} can appear more than once in E. An undirected graph in which no element of E appears more than p times is called a* p-undirected graph*.*

Similar to the case of (directed) graphs, also undirected graphs can be simple:

Definition 2.4 *A* simple undirected graph *G=(N,E) is a 1-undirected graph where*

$$\forall n \in N : \{n, n\} \notin E$$

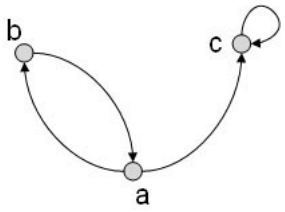

Figure 2.1: Visualization of the graph given in example 2.1

In addition to the different graph structures above (directed and undirected graphs), an extension that is important within this thesis is that of a hypergraph. This concept, which is a generalization of graphs, allows for edges that connect more than two vertices, and thus adds a considerable degree of expressiveness. Note that this definition as given by (Berge, 1976) allows for infinite edge families, but not for unconnected vertices.

Definition 2.5 *Let $N = \{n_1, n_2, ..., n_j\}$ be a finite set, and let $E = \{E_i | i \in I\}$ be a family of subsets of N (with I being an index set). Then H=(N,E) is called a* hypergraph *if*

- $\forall i \in I : E_i \neq \emptyset$, *and*
- $\bigcup_{i \in I} E_i = N$.

In analogy to the case of graphs, N is called the set of vertices, and the elements of E are called edges.

Example 2.2 *Let $N = \{a, b, c, d, e\}$, and $E = \{e_1, e_2, e_3, e_4\}$ with $e_1 = \{a, b\}, e_2 = \{a, b, c\}, e_3 = \{c, d\}$, and $e_4 = \{e\}$. Then H=(N,E) is a hypergraph with three vertices and four edges.*

It is worth noticing that in all these definitions (and in contrast to a naive interpretation of the term graph), graphical representations do not play any role - mathematical graphs are defined solely on an abstract level. As will be shown in the next part, most (yet, not all) results of graph theory are indeed not related to visual representations.

Throughout mathematical literature, however, a common and intuitive visualization of graphs is used: vertices are represented by dots and edges by lines or arrows, depending on whether they are directed or not. Figure 2.1 shows such a typical representation of the example graph 2.1, and figure 2.2 illustrates the hypergraph of example 2.2. The representation is of course not exactly determined by the original graph, as the latter does not contain any piece of information about visual aspects, such as position of nodes, shapes of edges, etc.

2.1.2 Important Results and Fields of Study

Based on the principal definitions as presented before, the mathematical theory of graphs covers a wide range of results. A full overview of these cannot be given within this thesis. However, it is reasonable to present some principal *areas* with significant contributions from graph theory:

Structure and Mappings. Based on the core definitions, some characteristics of graphs are made explicit in definitions. Examples include the order of a

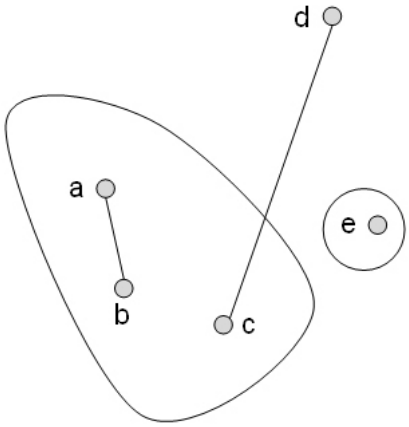

Figure 2.2: Visualization of the hypergraph given in example 2.2

graph (which is the amount of its vertices), or specific indices that measure the connectivity of graphs. Furthermore, graph theory characterizes mappings between graphs that retain the internal structure (graph isomorphisms), and formalizes substructure relations for graphs. Even in these basic areas, not all questions are solved from a computational point of view: for the isomorphism problem of two graphs, e.g., a final result concerning its complexity is not yet available - the problem is neither known to be in P, nor proved to be NP-complete (Skiena, 1990).

Paths and Circles. A path in a graph is defined as a sequence of edges. The existence of paths between vertices, the distance of vertices in graphs, and the handling of cycles (paths from a vertex to itself) are subject of graph theory. An important subtopic with lots of applications is the treatment of connected graphs without circles, named trees. These structures have found a lot of application areas in computer science, as they represent any kind of hierarchical structure nicely.

Flows in Networks. Adding a capacity attribute to edges allows the analysis of possible flows in the resulting network. Here, an important question is the calculation of a maximum flow, e.g. in a tube system.

Graph Traversals. Two essential and related questions motivate this area. The first is the question of finding a path in a graph that contains each edge exactly once - an Eulerian tour. The second one is dual to this and requires a path that contains each vertex exactly once, a so-called Hamiltonian path. Several other known questions like the "postman problem" or the "traveling salesman problem" base on these two.

Planarity. The conditions that have to hold in order to be able to imbed a graph representation in some surface without having intersecting edges is subject of research. For the case of planar imbedments, this has been completely solved and reduces to determining whether the graph contains one out of two specific subgraphs.

Colorings. Originating from the question whether any geographic map can be colored with four colors so that no two adjacent countries have the same color, a subdiscipline of graph theory concerned with attributing vertices and

assuring that any two neighbor vertices have different attribute values, has developed. There is still no closed formula that gives the minimum number of needed values to ensure this property for an arbitrary graph. For the case of graphs that allow for an imbedding in the plane, this question is solved - and even caught the attention of a general public (Wilson, 2002).

Not surprisingly, most mathematical contributions in the field of graph theory are on a conceptual and structural level, and do not relate to the representation of graphs. Two exceptions to this are the aspects of planarity and colorings - however, also here, the used methods are algebraic and "only" the results can be applied to visual representations.

Beyond the theoretical and conceptual results that graph theory provides, in particular the latter five mentioned areas also offer *algorithmic* results that make explicit how to find solutions to certain problems. Prominent examples include an algorithm to determine the shortest paths in graphs (Dijkstra, 1959), and a method to calculate the maximum flow in networks (Ford & Fulkerson, 1962). These more process oriented results constitute an important connection between mathematical theory and process-oriented approaches in computer science.

2.1.3 Discussion

As outlined, the fields of graph theory are manifold. Clearly, not all results will be directly applicable to the implementations within this thesis.

I see the primary contribution of graph theory for the developments within this thesis in the area of *conceptualization*: the formal definitions of graphs, vertices, edges and their relations constitute a solid foundation for extensions towards functionally enriched and typed graph structures, the latter seemingly needed in the context of using graphs for modeling.

The *structural* results of graph theory, in particular the mappings between graphs and the subgraph relation, are also interesting. Yet, as these are further investigated and presented in a computationally more accessible way in the domain of visual languages (cf. section 2.2), the impact on my work is only indirect.

Finally, some *algorithmic* results, especially around questions of path finding in graphs, are worth considering, particularly on the level of concrete modeling and simulation languages, but also for synchronization of graph based models (cf. chapter 4).

2.2 Visual Language Theory

As shown in the previous subsection, the mathematical theory of graphs allows for detailed insight into structural characteristics of graphs, understood as abstract constructs that consist of vertices and edges. However, this theory does not address or even systematically formalize visual graph representations. This is on the research agenda in the field of *visual languages*, which investigates visual representations (not restricted to graph representations!) with their syntax, semantics, and their use for communication - all connotations of the word "language". A concrete and broadly accepted definition of the term visual language did not yet evolve. Costagliola, Delucia, Orefice, and Polese (2002) have offered the following:

> *"A visual language may be conceived as a collection of visual sentences given by graphical objects in the two- or higher-dimensional space. Syntax of visual languages is described through the graphical objects of the language (the vocabulary), the relations used to compose the sentences,*

> *and a set of rules defining the visual sentences belonging to the language. The graphical objects of a visual language are characterized by a set of attributes that can be classified as graphical attributes, syntactic attributes, and semantic attributes."* (page 575)

A lot of contributions in the research area of visual languages are application oriented in that they investigate in the usage of specific visual languages for certain domains. Within her research on the use of visual languages for discussion support, Gaßner (2003) has identified several following different *aims* for the use of visual languages, including reasoning in visual representations, learning support, knowledge elicitation, and the use as communication means and knowledge product. Her points stress the potential of considering visual language theory as a foundation for my collaborative modeling framework implementations.

The following parts of this section describe different theoretical approaches in the field of visual languages. I will give an overview on current methods that formally and systematically handle visual languages, and discuss implications for the work within this thesis.

2.2.1 Classification Schemes for Visual Languages

Given that the target for this thesis is a framework to support multiple integrated representations, it is reasonable to search for visual language classification schemes. Here, a literature review yields essentially three different approaches.

The idea of the first one, given by Costagliola et al. (2002), is to take the modality by which visual sentences are composed as discriminating factor between different *classes* of visual languages. They identify three basic classes: *connection-based* languages, in which visual sentences are formed by interconnecting graphical objects, *geometry-based* languages, where sentences are created by spatially arranging graphical objects, and *hybrid* languages. In the latter case, both interconnections and spatial arrangement are of importance. For the former two classes, the authors present the following subclass hierarchy:

Plex. The plex class is connection-based and consists of graphical objects that only have a limited number of connections, which can be added at *attaching points*.

Graph. The graph class is the super class of the plex class and allows for an unlimited number of connections to a set of *attaching regions*.

String. This class is geometry-based and represents the reduction of visual languages to the textual case. Graphical objects of this class are textual characters - the 2-dimensionality comes from their visual representation, not their information type.

Iconic. This generalization of the string class allows for icons, which are defined as images within boxes of equal fixed size.

Box. Members of this super class of the iconic class are characterized by their rectangular bounding box, which can be of variable size.

Table 2.1 contains syntactic attributes and typical relations for each of these classes, together with example languages that belong to the classes. Commenting their results, Costagliola et al. (2002) state that many modeling languages mix elements from different subclasses, in particular within one major class (e.g., plex objects with graph objects). They also mention the UML (Booch et al., 1998) as an important hybrid modeling language - state chart diagrams or package diagrams, which make use of connections as well as spatial arrangements, illustrate this.

Table 2.1: Classes of visual languages

Class	Attributes	Relations	Example
Plex	attaching points	plex interconnections	Flowcharts
Graph	attaching regions	graph interconnections	Petri Nets
String	position of character in string	string concatenation	Texts
Iconic	position of icon in grid	spatial concatenation	Puzzles
Box	position and size	complex spatial relations	State charts

Marriott and Meyer (1998) take a completely different approach in their classification of visual languages. They provide a systematic hierarchy of visual languages based on formal properties, similar to the case of the Chomsky hierarchy for text based languages (Chomsky, 1959). This way, they achieve a characterization of visual languages based on their fundamental computational properties. Their primary means base on the following grammar definition:

Definition 2.6 *A* constraint multiset grammar *(CMG) over a computational domain D is a quadruple (T_T, T_{NT}, T_S, P) consisting of a set of terminal symbols T_T, a set of non-terminal symbols T_{NT} with $T_T \cap T_{NT} = \emptyset$, a start symbol $T_S \in T_{NT}$, and a set of productions P. All symbols except for the start symbol can have attributes. A production $p \in P$ has the following form:*

$$xX'_1 \dots X'_m \rightarrow X_1 \dots X_n X'_1 \dots X'_m \| C \wedge \overrightarrow{v} = E$$

Here, x is a non-terminal symbol, and $\overrightarrow{v}$ are the attributes of x. $X_1, \dots, X_n$ and $X'_1, \dots, X'_m$ are terminal or non-terminal symbols with $n \geq 0$, and the constraint C and the expression E are over the attributes of $X_1, \dots, X_n$ and $X'_1, \dots, X'_m$.

Derivation steps are defined in analogy to text grammars, with the additional step of checking the constraints and assigning new attribute values as resulting from specific computations. Marriott and Meyer (1998) state that the CMG class is too powerful to allow the construction of a reasonable hierarchy, due to the arbitrary calculation of attribute values. As a consequence, they restrict the expressiveness of CMGs by allowing only attribute copying instead of calculating. The severeness of this step becomes clear upon consideration that the attributes are the elements that really contain the visual information in the language - typical attributes relate, e.g., to positions or connections of elements.

The resulting class of grammars, *copy-restricted constraint multiset grammars* (CCMGs), is further subdivided into several types (similar to the Chomsky hierarchy), and results in a hierarchy of the corresponding languages.

Marriott and Meyer (1998) show the equivalence of CCMGs with other grammar types (cf. next subsection), and analyze the complexity of the membership problem (given a CCMG G and a sentence S, can S be generated by G?). One important result is that even for the most restricted type of CCMGs, the membership problem is NP-complete. In the general case, it is even undecidable.

The approach of Bottoni, Costabile, Levialdi, and Mussio (1998) classifies visual languages with respect to human computer interaction criteria. They take a pixel

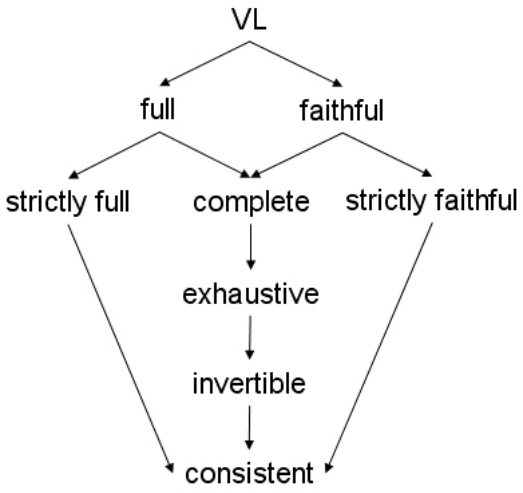

Figure 2.3: Inclusion relations for visual language classes in the classification of Bottoni et al.

oriented approach to formalize images and image descriptions over attributed symbols, and define a *visual sentence* essentially as a triple $\langle i, d, \langle int, mat \rangle\rangle$, consisting of an image i, a description d, an interpretation function int and a materialization function mat. int essentially maps images to descriptions, mat maps in the other direction. A *visual language* is defined as a set of visual sentences.

Based on these low level definitions and properties of the int and mat functions, Bottoni et al. (1998) give a taxonomy for visual languages whose relationships are illustrated in figure 2.3.

The most important classes are the following: *strictly full* languages have no ambiguities in image interpretation, so that a characteristic structure is always interpreted in the same way, even if it may appear in different contexts. A language is *strictly faithful* if a given description is always materialized in the same characteristic structure, i.e. no multiple representations of the same description are possible. *Invertible* languages ensure that each pixel in the image is controlled by exactly one system component, and that a click on that pixel allows the user to refer to that component. Finally, *consistent* visual languages are those that are strictly full, strictly faithful and invertible.

Bottoni et al. (1998) study systems that belong to the different classes of their taxonomy, and derive interaction characteristics of these systems.

2.2.2 Formal Approaches and Employed Methods

The methods employed in visual language research with the aim of specifying and recognizing visual languages are heterogeneous. According to Marriott, Meyer, and Wittenburg (1998), we can identify three different general lines, which rely on either grammars, logic, or algebra.

Grammar Based Approaches

The grammatical approach is based on mechanisms similar to the ones used in string language specification. It is a constructive method that is useful to formalize the notion of "belonging to a language" and that consequently has potential for parsing applications. A wide variety of grammar based approaches exists, each

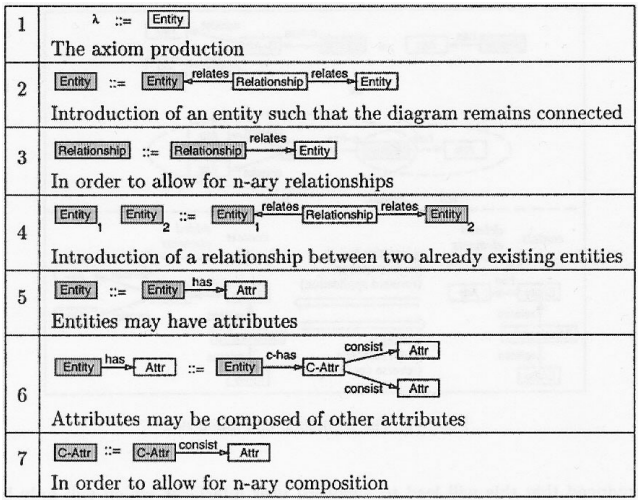

Figure 2.4: Example graph grammar for Entity-Relationship diagrams

with specific strengths concerning expressiveness, suitability for specific notations, and options for efficient parsing. Marriott et al. (1998) give a good survey on the existing state-of-art in this area. One example, the (C)CMG approach, has already been described in the previous subsection. For the topic of this thesis, two other important grammar types are worth considering:

Graph grammars use graph structures, usually enriched with attributed vertices and/or edges, as sentences and (consequently) within production rules. There is a variety of different graph grammar formalisms, e.g. precedence graph grammars (Kaul, 1982), or edNLC (edge-labeled directed node-label controlled) graph grammars (Brandenburg, 1988). Rekers and Schürr (1997) give a very intuitive definition of a graph grammar in a general sense. They rely on the mathematical notion of graphs, enriched with labels for edges and vertices:

Definition 2.7 *A* graph grammar *G is a tuple (A,P), with A a nonempty initial graph (the axiom), and P a set of graph grammar productions. A production $p \in P$ is a tuple (L,R) of graphs over the same alphabets L_V and L_E of vertex and edge labels. It can be applied to a graph G and rewrites it into $G' = G \cup R \setminus M_L$, if G contains a subgraph M_L that matches L. The set $K := L \cap R$, is called the* context *of a production.*

An example set of productions for abstract Entity Relationship syntax graphs is shown in figure 2.4, which is taken from Rekers and Schürr (1997). Even though the figure just shows a visual representation of the productions (and not their formal notation), the intuitive character of graph grammars becomes clear.

The second class of grammars I want to mention are *relation grammars and relational grammars*. These approaches rely basically on identifiable entities and typed relations that may exist between these entities. The grammar allows for derivation steps in these structures. Even though they are very similar to graph grammars concerning expressiveness (Marriott & Meyer, 1998), they put a different focus by not considering graphs as a whole, but emphasizing on the entities, the latter being in certain relations to each other. This way, relation grammars are targeted at a slightly wider range than graph grammars. It is, e.g., relatively easy to express visual relations like "A is x-centered to B", or "X is below Y". The concrete inclusion mechanism for the relations in the grammar varies. Wittenburg and

Weitzmann (1998),e.g., rely on constraints and attributes. Ferrucci, Tortora, Tucci, and Vitiello (1998), on the other hand, have terminal and non-terminal symbols for relations: in their *Symbol Relation grammar* approach, they separate the symbol productions from the relation productions. A derivation step is subdivided into two phases, which distinguish between element rewriting and relation treatment: After the application of a symbol rule, the parser looks for the corresponding relation rules (the correspondence is explicit in the grammar), and applies them.

Logic Based Techniques

The previous part outlined how computational grammars can be used to define visual languages. One strength of that approach is that it principally allows for recognizing and producing sentences of languages. A disadvantage is that most grammar based techniques treat the visual information as attributes of specific elements, usually (for reasons of flexibility) with an unrestricted set of possible attribute names. This implies that visual relationships, apart from element interconnections, cannot be dealt with in much detail. Thus, geometric-based languages and hybrid languages in the sense of Costagliola et al. (2002) are not really adequately covered.

Here, logic based techniques play an important role. These usually rely on spatial logics which axiomatize different possible topological (geometric) relationships between objects, and apply a certain reasoning mechanism which operates on the basic relationships.

Two prominent examples for the sets of axioms are the Region Connection Calculus (RCC) presented by Randell, Cui, and Cohn (1992), and the Cardinal Direction Framework (Ligozat, 1998).

The RCC logics bases on one primitive reflexive and symmetric relation $C(x, y)$, which holds if the topological regions x and y are connected in the sense that they have at least one point in common. RCC uses this relation to define the following eight basic relations, which are illustrated in figure 2.5:

- DC: disconnected
- EC: externally connected
- PO: properly overlaps
- TPP: tangential proper part
- TPP^{-1}: have as tangential proper part
- $NTPP$: non-tangential proper part
- $NTPP^{-1}$: have as tangential proper part
- EQ: equals

As visible, RCC does not use the notion of directions or orientations. Several other approaches do, e.g. by using angles, qualitative orientations, or cardinal directions. The latter has been investigated in depth by Ligozat (1998). His basic relations as illustrated in figure 2.6 are very intuitive. Recent developments like, e.g., the SpaceML approach presented by Cristani and Cohn (2002), aim at integrating different axiom sets in order to extend the expressiveness of the resulting relations.

Similar to the heterogeneity of the atomic relations sets, also the applied reasoning methods vary and cover a broad spectrum of established logic calculi. Of course, issues around decidability and efficiency play an important role here. For these reasons, we can frequently observe the use of a restricted first order predicate logic as a foundation.

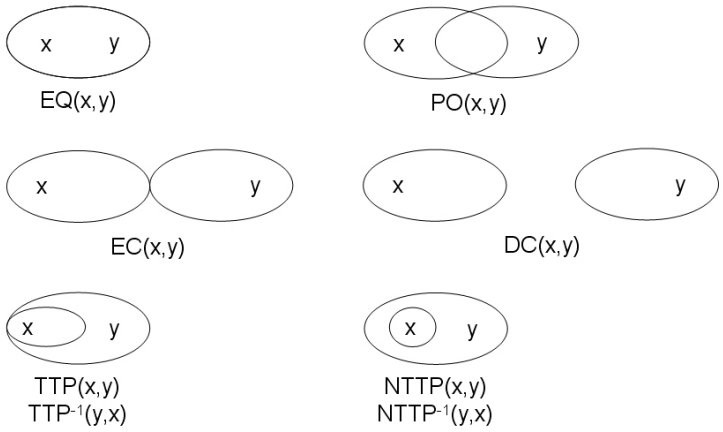

Figure 2.5: The RCC basic relations

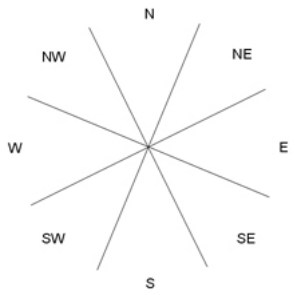

Figure 2.6: The basic relations of the Cardinal Direction Framework

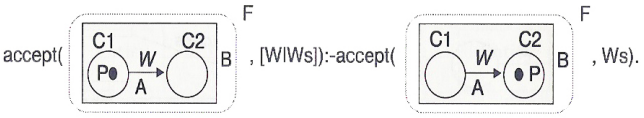

Figure 2.7: A picture logic rule

Some of the existing techniques make use of *definite clauses*, which have a quite close connection to context-free grammars and allow for a straightforward implementation in Prolog. Here, an interesting example is the work of Meyer (1994) who extends this idea by integrating visual expressions into logic programming. His basic idea is to introduce the notion of a picture term, which has visual constants and visual variables. The unification mechanism of the logic programming framework is extended to a picture unification, which allows for using picture terms wherever normal terms can be used. Figure 2.7, taken from Marriott et al. (1998), shows an example of such a rule, which nicely illustrates how the illusion of operational semantics can be achieved with picture logics.

The approach of *constraint logic programming* integrates logic programming with constraints in some given (usually mathematic) domain. It is also a suitable method for visual language formalization, as it generically allows for, e.g., coordinate calculations. Furthermore, it is possible to use constraint logic programs to build parsers for CMGs (cf. definition 2.6).

Some reasoning systems rely on variants of description logic, a theory which is based on the idea of structured inheritance networks. Practically usable description logic based approaches cover a subset of first order predicate logic and offer a complete reasoning system, i.e. they are decidable. A prominent example for the use of description logic is the work of Haarslev (1999). His $\mathcal{ALCRP}(\mathcal{D})$ formalism is domain independent in the sense that it allows for concept and role (binary relations) definitions, and offers a complete reasoning service.

Algebraic Methods

Loosely speaking, the logic based approaches as presented in the previous part exploit visual relations to make meaning. Other more algebraically oriented methods rely more on structural information, and try to describe complex visual structures by defining atomic elements and composition functions.

Of course, also some logic or grammar oriented approaches use algebraic structures (e.g., the RCC basic relations). In addition, "pure" algebraic techniques are rare in the field of visual languages, as usually some procedural parts (e.g., for structure recognition) are addressed. In this sense, there is no sharp distinction between the different categories. However, there are some methods that indeed put a strong emphasis on type theoretic concepts and structural relationships, which justifies their classification as "algebraic".

In this area, the work of Wang and Zeevat (1998) is a good example. They present a syntax based algebraic approach which allows for semantic reasoning about pictures. The central purpose of their work is to define meaning of pictures in terms of one domain, and then derive the meaning of pictures in another domain by using an analogy relation. The basic concepts they provide are the following:

Definition 2.8 *A* graphical signature Σ *is a quadruple* $(\mathcal{S}, \leq, \mathcal{F}, \mathcal{P})$*, where:*

- $\mathcal{S}$ *is a set of graphical sorts to which graphical objects belong,*
- $\leq$ *defines a partial order on* $\mathcal{S}$ *and expresses subsort relations,*

- $\mathcal{F}$ *are operations over graphical objects. Wang and Zeevat (1998) distinguish between constant functions, natural functions, artificial functions, and attribute functions,*
- $\mathcal{P}$ *contains graphical predicates.*

In addition to these syntactic specifications, *graphical theories* are used to specify geometrical properties shared by all the pictures in the language:

Definition 2.9 *A* graphical theory *over a graphical signature* Σ *is a set of formulas over* Σ *which is closed and consistent under the consequence relation of the underlying logical system.*

Based on these concepts, Wang and Zeevat define partial homeomorphisms between graphical and domain signatures and thus establish connections between interpretation of graphical structures and meaning in underlying application domain signatures.

Another algebraic technique that does not primarily target visual structures, but interaction with these structures, has been proposed by Dinesh and Üsküdarlı (1998). They use a specification language to define visual objects, define the syntax of a visual language that makes use of these atomic objects in a context-free grammar, and describe language semantics through conditional equations. The rewrite engine they use for parsing structures considers dynamic input and output in the sense that it allows for "holes" in terms. Once such a hole is encountered, the engine stops and asks for further input. The framework is then expected to get this input by asking the user for additional specifications. The repetition of this process thus allows for interactive diagram construction.

2.2.3 Discussion

As presented in the previous subsections, there are a lot of constructive approaches in visual language theory that aim at providing theoretical frameworks for visual languages together with algorithmic techniques that allow the handling of sentences in these languages. All these approaches share a common problem: visual structures, even restricted to graph structures, are inherently complex so that a large number of "interesting" questions are NP complete or even undecidable - a prominent example relevant within this thesis is the membership problem ("does an expression belong to a visual language?"), which has applications in the fields of model syntax and also semantics checks. In visual language theory, the complexity problem is usually "solved" by restricting the scope of covered languages in a way that allows for an efficient handling of the resulting set. Examples are the limitation of grammar productions (Rekers & Schürr, 1997) or the restriction of attribute calculation to value copying (Marriott & Meyer, 1998).

All these restrictions are in conflict with some of the core purposes within this thesis, namely with the aim of building a framework that supports heterogeneous and flexible graph structures. A restriction of the supported primitive elements that goes beyond the restriction to graph structures is problematic, as it limits the expressiveness of the whole approach. In addition, this limitation does not really seem to be *required* within the purposes of this thesis. Considering the targeted interactive and constructive usage of the modeling tool, it cannot be expected that all intermediate steps in model construction always constitute a "correct" model in the classical sense that it is a member of some language of correct sentences, and therefore can be accepted by some grammar. Due to this, the methodology and technology within this thesis will have to support also structures which do only fulfil minimal syntactic correctness constraints. Another reason for not relying on

a central method for ensuring correctness on a higher semantic level (e.g., using grammar based formalisms) is that I explicitly intend to support heterogeneous models, which consist of mixed interoperable structures. Here, a central definition of correctness is hard to give, especially on the semantics level: under which conditions, e.g., is a Petri Net that is annotated with a concept map, semantically *correct*?

Despite this general difference between the approaches in the area of visual language theory and the particular aims within this thesis, some results are worth considering. The *formalization* of visual structures as objects with attributes and relations like, e.g., done in the approach of Wang and Zeevat (1998), is an important result. It is not sufficient though, because it does not cover interactive structures that go beyond object attributes by allowing operational features - which is what is needed for modeling applications, but usually not considered in visual language research.

The notion of constraints as contained in a lot of the mentioned approaches is also considerable. Even though I do not plan to check heterogeneous graph structures for correctness in a general sense, some method to guarantee at least a minimal syntactic integrity will be needed in order to allow for an interpretation at all.

Apart from these impacts from visual language research on the core work within this thesis, most of the presented results have applications in related fields. The mentioned logic formalisms and grammar based parsing mechanisms can play an important role, e.g., when trying to decide if a specific graph structure is correct in a higher, task-related, sense - an issue which is highly relevant within tutoring applications.

2.3 Meta Modeling

As motivated in the introduction chapter, core parts of the implementations within this thesis are targeted towards a flexible and interoperable modeling framework. Taking this into account, existing techniques that generically describe modeling languages and their relations are of high importance. Here, a literature review yields three different research directions: some approaches aim primarily at *describing* modeling languages, others offer *transformative* techniques that allow for language switches. Finally, a small number of papers address *interoperability* issues in integrated models. The following subsections describe all three directions.

2.3.1 Descriptive Frameworks

All flexible modeling frameworks that are able to deal with multiple modeling languages usually employ some kind of metastructure that describes the different languages and, potentially, also their connections. These metastructures are often used implicitly, only few authors explicitly declare their frameworks as e.g. meta modeling environments (Lara & Vangheluwe, 2004).

Obviously, both the level of detail and also the focus of meta modeling techniques can vary considerably, and a standardization of these frameworks is of course an issue worth investigating, as it promises reusability and interoperability on the level of both models and modeling languages. Yet, this is difficult due to the heterogeneity of modeling languages and the critical question how to cope with things beyond syntax: model semantics and dynamic functional issues are, e.g., necessary topics to address in a full-scale solution.

Within this subsection, I want to outline two prominent descriptive meta modeling techniques which are representative for the current state-of-art in this area.

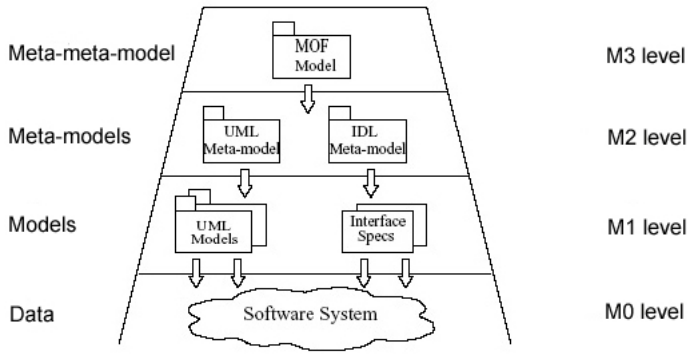

Figure 2.8: The MOF architecture with example meta models

Meta Object Facility

The first example is the Meta Object Facility MOF (Meta-Object Facility Specification, n.d.; Distributed Systems Technology Centre, n.d.), which is the OMG standard for defining, representing and managing metadata. Among other things, the OMG MOF and associated specifications define the *MOF Model* as the MOF standard abstract specification language for meta-models that define different kinds of metadata. The specification includes an associated abstract mapping that relates a meta model to the corresponding information model for metadata, i.e. it says what a meta model means from the information perspective.

The meaning of these things become clear with the general conceptual layer architecture that the MOF specification proposes. This consists of four layers: the MOF Model itself, meta models, models, and data. Figure 2.8 shows these different layers for the example of an instantiation that handles metadata representations of UML models and CORBA IDL.

The top (M3) level of the framework consists of the standard MOF Model, i.e. the standard abstract language for defining MOF meta-models. This is the fixed point that unifies the MOF metadata architecture.

On the M2 level, the conceptual framework contains the meta models for the metadata that is required by the application. In the example case, these are the UML meta model and the specification of CORBA IDL. It is important to note that the presence of these two meta models does not say anything about their relation in the application: in the example, IDL specifications might be generated from UML diagrams, or they might be created by the user and linked to UML diagrams. Even the extreme case - an isolation between UML and CORBA models - is not prohibited.

The M1 level of the architecture contains models expressed in conformity to the meta models contained in level M2. In the example case, these are UML diagrams and CORBA interface specifications of a system. In the MOF framework, models on the M1 level would typically be produced by the user using tools in the software development environment.

The lowest level M0 of the framework contains the software system. Depending on the scope of the application and the example metamodels involved, one might expect to find IDL and program language source files, and binaries.

The typical use of the MOF framework in a development process consists of the following four steps (Distributed Systems Technology Centre, n.d.):

Informal Diagram Definition. The first step in developing a MOF based solution consists of defining the MOF meta-models for the metadata framework.

Table 2.2: The core concepts of MODL

Name	Main Properties
Class	Inheritance information, abstraction degree, attributes with name, type, multiplicity and scope
Association	Two ends, each with name, class type, multiplicity, and aggregation type
DataType	Data type without object identity, several predefined primitive types are specified.
Package	Other meta model elements defined within the package, e.g. Classes, Associations, DataTypes
Constraint	Name, constrained element, constraint expression (usually in OCL), evaluation policy
Reference	Referenced class type
Operation	Parameters and raised exceptions
Exception	Name and parameters
Constant	Name and simple value
Tag	None (mechanism for tailoring and extending the meta modeling language)

The developer will typically start by drawing an informal meta model diagram (using a convenient modeling language like, e.g., UML).

Specification in MODL. In order to be useful within the MOF framework, the informal specification must be transformed to a formal meta model definition in the text based MODL language. As the format in the first step is unspecified, this step will typically have to be done by hand. The MODL language syntax, defined formally through a grammar, resembles the IDL syntax and contains concepts that are closely related to modern object oriented programming languages, as table 2.2 shows.

Repository inclusion. Having specified the meta model formally, the next step in the MOF methodology is to generate a repository that can manage metadata conforming to the meta model. MOF also contains a specification of a repository structure for meta models.

Application construction. The MOF method foresees the implementation of tools that make use of the repositories as the final step. To simplify and standardize this, the formal MODL specification can be exploited, e.g. through automatic IDL mappings that can serve as starting points for the implementations.

Domain-Specific Modeling

As outlined, the essence of the Meta Object Facility consists of a very general high level framework for meta modeling and the text based language description language MODL. Other meta modeling techniques take a more development oriented perspective and aim primarily at facilitating the steps from modeling to system construction. A good example for these approaches is Domain-Specific Modeling (DSM) (Metacase, n.d.). The basic motivation that underlies this modeling suite is that common modeling languages like UML contribute surprisingly little to the process of accelerating and improving software development, because a solution for

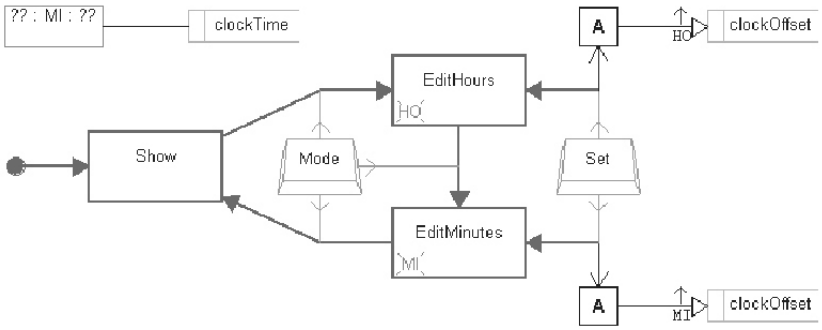

Figure 2.9: Time setting modeled in DSM

a domain problem first has to be solved in the domain with little or no tool support, then this solution has to be mapped to UML. From there, some code generation can be done, however there still remains a lot of manual programming work (Metacase, n.d.). The core idea of DSM is to reduce the resource-intensive and error-prone mappings between representations, and to develop a solution for a domain problem only once and in a domain specific language that the user can easily handle. From this language, a direct translation to (nearly fully functional) program code is aimed at. The advantages of this procedure, as argued by Tolvanen and Kelly (2004), are the following:

- Having specifications on a significantly higher level of abstraction than traditional code or class diagrams means less work for developing the specifications.
- DSM reduces the need to learn new semantics on the user side - the conceptualizations of the problem domain are usually known and conceived as "natural", in contrast to external notations like UML.
- Specifications in domain terminology are usually better understandable and superior for memorizing, validating, and as a means of discussion.

Figure 2.9, taken from (Metacase, n.d.), shows an example for a model in DSM. The model, which describes the process of setting a clock with two buttons, is designed to be intuitively understandable for those people that have the appropriate domain knowledge. In order to make use of DSM models in the intended way, three steps are necessary (Metacase, n.d.):

1. a modeling language that fits the domain,
2. a tool for building models in that language, and
3. automatic code generation for models in the language

The first of these steps includes the definition of the following three elements:

Domain concepts. These are taken directly from the problem domain and are given properties typical for the domain.

Notations. A graphical visualization for the domain concepts, typically developed with end user participation.

Rules. The guarantee that all models are "correct" in domain terms. The rules can be of different kinds and relate, e.g., to associated concepts, or the layering of models. The reuse of designs for rules is explicitly foreseen.

While the second step (the tool for building models in the supported language) can be generically supported by metamodeling tools for DSM models like Metacase (n.d.), the code generation is of course (together with the domain rule specification) the most problematic of the three. Tolvanen and Kelly (2004) admit that for this step a lot of manual programming work is needed. Yet, they argue that this work usually has to be done only once for a domain - as typically a user shares the same domain model for a lot of applications, this single manual step is not really a disadvantage. The code generating components can be reused in the same way as the domain concepts that they refer to.

As presented, it seems that DSM is no "real" metamodeling language, as it does not define an explicit format for the description of models. In fact, this nonexistence (or: hiding) is done on purpose and with the aim of simplicity. Despite this, it makes definitely sense to classify DSM as a metamodeling technique, as it is an approach that indeed covers a number of modeling languages. The DSM metamodel is only implicitly given and can be conceived as a combination of the hidden typology that connects the user defined concepts and properties, the primitives that can be used for defining notations and rules, the mechanisms that link concepts to notations, parts of the code mapping, and the connection features for models.

2.3.2 Transformation Approaches

There are a number of research results that contain methodologies for combining multiple modeling languages with the aim of *transforming* one model (specified in a modeling language) to another model, which can either belong to the same language, or to a different one.

Simple Attributed Graph Transformations

As a lot of modeling languages employ objects and relationships between these objects as primitives, it is not surprising that also on the level of model transformations, there are several approaches that explicitly focus on graph structure transformations. As already noted in section 2.2.2, one possible solution are graph grammars.

Lara and Vangheluwe (2004) make use of these grammars for attributed graphs not only for transformations, but also to support the execution and optimization of models. In their approach, model execution ("simulation") is essentially the modification of graph attributes according to a grammar, whereas optimization steps perform structural changes in the graph structure with the aim of reducing complexity and improving performance. They propose an implementation of a graph grammar engine which allows for several execution modes (step-by-step, continuous, and animated), and considers element-subtype relationships. The latter allows for reusing transformation rules in different contexts.

Transformation approaches on graphs generally have to face that a lot of problems in the area of mappings between graphs are computationally hard, mostly at least NP complete, so that efficient general-purpose algorithms seem out of reach. One critical problem that belongs to this category is the determination whether a given graph structure can be obtained from another one by the means of a set of transformations. Here, a subproblem is the matching of two graphs. Cordella, Foggia, Sansone, and Vento (1998) try to solve this problem using subgraph transformations. Their work uses feasibility rules and gives *inexact* matchings that only approximate a final solution. They make use of transformations in order to iteratively search for a matching. The basic types of transitions are split, merge, delete branch, and insert branch.

Hypergraph Based Schema Transformations

A formalization of a transformation technique for the case of hypergraph based data structures has been proposed by McBrien and Poulovassilis (1999). Their primary notion is a *schema*, which is a structure that consists of a labeled, directed, and nested (i.e. edges can connect both vertices and also other edges) hypergraph, and a set of constraints, the latter being boolean valued queries over the hypergraph. An *instance* of a schema is defined as follows:

Definition 2.10 *For a schema* $S = \langle Nodes, Edges, Constraints \rangle$*, an* instance I *is a set of sets satisfying the following:*

- *Each construct* $c \in Nodes \cup Edges$ *has an extent, denoted by* $Ext_{S,I}$*, that can be derived from* I*,*
- *Conversely, each set in* I *can be derived from the set of extents* $\{Ext_{S,I}(c) | c \in Nodes \cup Edges\}$*,*
- *For each* $e \in Edges$*,*$Ext_{S,I}(e)$ *contains only values that appear within the extents on the constructs linked by* e *(domain integrity),*
- *The value of every constraint* $c \in Constraints$ *is true, the value of a query* q *being given by* $q[c_1/Ext_{S,I}(c_1), \ldots, c_n/Ext_{S,I}(c_n)]$ *where* $c_1, \ldots, c_n$ *are the constructs in* $Nodes \cup Edges$*.*

Finally, a *model* in the sense of McBrien and Poulovassilis (1999) is a triple $\langle S, I, Ext_{S,I} \rangle$. Based on these formalized concepts, they define the following primitive transformations on models:

- renaming nodes and edges,
- adding and removing constraints,
- adding and removing single nodes,
- adding and removing single edges

For the nodes and edge transformations, the approach uses queries to identify the entities to be added or removed (in contrast to other techniques that rely on unique identifications of elements).

Sequences of these primitive transformations are called *composite transformations*, these correspond to transitions between models within one modeling language. Based on these, McBrien and Poulovassilis (1999) also analyze specific requirements for mapping "richer semantic modeling languages" and point out their ideas using the example of UML and Entity Relationship models.

Primitive and composite transformations allow for intra-model mappings in the sense that the syntax of the concrete modeling language limits the mappings. Beyond that, McBrien and Poulovassilis also aim at supporting inter-model transformations. Here, their basic idea relies on defining primitive add and delete transformations, where the extent of a construct which belongs to a modeling language M_1 is defined in terms of the extents of constructs in some other modeling language M_2. Based on this, McBrien and Poulovassilis state that generic template transformations which automatically translate between different languages are possible within their approach. They list the following two important criteria for such generic inter-model transformations:

1. Ensure that every possible instance of a construct in M_1 appears in the query part of a transformation that adds a construct to M_2.

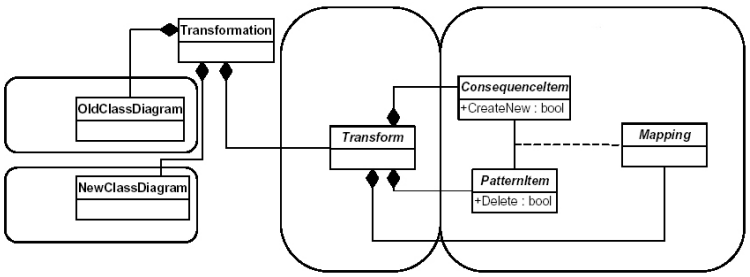

Figure 2.10: Structure of domain evolution tools

2. Ensure that every construct c of M_1 appears in a transformation that deletes c.

It is worth noting that McBrien and Poulovassilis treat the intermediary constructs that neither completely belong to M_1 nor to M_2 (but that definitely occur in the process of inter-model transformations) as purely temporary objects: they do not state how, e.g., partially "translated" models (thus, mixed structures of M_1 and M_2), could be generically supported in their approach.

Higher Level Model Migration Techniques

Apart from the low level transformative approaches listed above, there are also some research results which put the focus on higher level model mappings. These have a natural relation to the meta modeling techniques as noted in subsection 2.3.1 in that they use meta modeling frameworks as a base for the transformations.

The mechanism proposed by Celms, Kalnins, and Lace (2003) allows multiple representations of a domain model in several modeling languages. Strictly speaking, they do therefore not directly address model transformations but view transformations - yet, it can be argued that multiple representations already constitute a specific kind of model transformation, as each language may display or hide different parts of the general domain model.

Celms et al. do not directly define mappings between different elements of languages, but take the MOF meta modeling framework (Meta-Object Facility Specification, n.d.) as a foundation and express both domain models and presentation models using this framework. They make use of bijective mapping associations between these two types of models, which allows them to maintain multiple domain model representations simultaneously, and to propagate changes in one representation to the others.

The work of Sprinkle and Karsai (2003) is in the area of domain-specific modeling languages (DSMLs). They address the problem of modeling language definitions that change over time, and propose a model migration process to incorporate these domain evolutions. In their approach, a DSML specific *domain evolution tool* handles metamodel differences, different domain evolution tools are managed within a *domain evolution framework*. Figure 2.10 shows the structure that the framework imposes on supported model migration tools. The approach essentially relies on *Transform* operators, which are the ingredients of any model migration algorithm. Transform operators can have several types (rules, tests, and cases), some of which may be sequenced. Any Transform operator relates to a set of *legal items*, which can be patterns (used to identify matching input components), or consequences (objects within the new DSML). In addition, a Transform operator also refers to a mapping association, which can, e.g., be a replacement, an insertion, or a concatenation.

2.3.3 Formal Interoperability Approaches

In addition to the mentioned lines of research which *describe* modeling languages in detail and that offer *transformative* mechanisms between models and modeling languages, there are some approaches that put the emphasis on *interoperability* and formal model integration. Obviously, existing state-of-art in this field is of high importance within this thesis.

Typologies of Integration Approaches

The aim of model interoperability is relatively clear: the development of methodologies that connect models which are expressed in different (formal) languages. These solutions are able to increase the reusability of models, as models can be put in different contexts and still "work". They also facilitate modeling tasks through an enhanced choice of means of expression. However, due to the vast heterogeneity of modeling languages and the resulting complexity of the interoperability task, a universal solution has not yet evolved and can not be expected for the next future.

Several authors give classifications that outline strategies for (partial) integration solutions. The classification of Dolk and Kottemann (1993) describes two basic categories:

Definitional Integration. Approaches in this category produce a single new model that combines a set of given models. Here, important conditions are that the new model is represented with the languages of the input ones, and that also the semantics of the resulting model corresponds to that of the original ones.

Procedural Integration. These techniques are characterized by their property that they leave the original models as they are, and add a superstructure which is capable of coordinating the connections between the models.

In a way, definitional integration involves the logical linking of related model representations, whereas procedural integration concerns the linking of processes to form operators in an integrated model.

Dolk and Kottemann (1993) also consider organizational benefits that model integration may offer, and analyze implementation related issues for integrated modeling systems. The list already presented on page 19 contains the major requirements they identify. For each of these requirements, Dolk and Kottemann refer to relevant research results. However, they state that no current implementation fulfils these criteria, and the only constructive point in their paper is the proposition of a feature list for a hypothetical Communicating Structured Modeling Language (CSML), which they characterize as similar to a discrete event simulation programming language, but with hooks that allow for a dynamic inclusion of model schemas, solvers, and relational data. According to them, the minimum system features are these:

- basic structured programming constructs of sequence, selection, and iteration,
- demons,
- embedded Structured Modeling Language statements for model definition,
- parallel execution of processes,
- transformation operators to solver data structures (in the context of structured modeling techniques),

- embedded SQL statements for data manipulation (their approach is focused on relational database applications)

Another analysis of principal possibilities for model integration has been done by Wang, Yeo, and Poh (1998). They focus on the area of structured modeling and take an object oriented perspective on model integration. Based on a classification that considers homogeneity degrees between model schemas, their analysis results in the following different types of model integration:

Amalgamation. This homogeneous case describes models of the same schema being integrated: the interpretation mechanism (i.e. the solver, for the case of structured modeling) for the resulting model can be the same as for the input models.

Combination. This case also represents an integration of models that belong to a joint schema. Different to the amalgamation case, here the resulting model is expressed in another schema, thus a new interpretation mechanism is needed.

Concatenation. If heterogeneous model schemes are integrated, the resulting model is called a concatenated model. Wang et al. (1998) propose the interpretation for the integrated model to be done by a process that uses the interpreters of the original models.

Embedding. If the integrated model schemas are heterogeneous but one is a subset of the other, then the solver of the larger schema can be used and the integration process is called an embedding.

Along these types, Wang et al. (1998) discuss also implementation issues. However, these are limited in scope since the modeling language is restricted to structured modeling, and the integration approaches are, in addition, specific to the domain of distribution systems.

Mapping Based Techniques

A significant number of attempts for model interoperability uses mappings between different models as a primary means. In the typology of Dolk and Kottemann (1993), these fall into the category of procedural integration. Two contrary approaches to interconnect heterogeneous models have emerged here: either with or without a connecting superstructure. In the sequel, I describe representative examples for both categories.

Of course, general frameworks for the integration of models *without* any kind of mediating control component are hard to build, as the communication between models is not directly controllable in this architecture. Wang and Liu (2003) analyze the different interconnection options between sets of models, and derive certain conditions which have to hold in order to be able to build combined models. They summarize their approach this way:

> *"A complete model management must assist in the selection, linking, and execution of models. That needs a general framework for formalization of models. By input and output standardization and model rules, it may be done to link different models together to solve a complicated problem."* (page 36)

Indeed, the essence of their approach is very simple: they understand a model as a black box that consists of a set of input parameters and one output parameter. Based on this, they define a model combination relation that expresses which model

output parameters can serve as input for other models, and characterize a *complete model combination* relation by the criterion that for any input parameter (in) of a model, there is at least one model with an output parameter (out) so that $\langle in, out \rangle$ is contained in the combination relation.

Based on these definitions, Wang and Liu (2003) define a composite model $MM = (MI, RI)$ for a set of models M, a combination relation R, and an expected joint output O through the following criteria:

1. $MI \subseteq M \wedge RI \subseteq R$
2. $\exists_1 m_t \in MI : out(m_t) = O \wedge (\forall m \in MI : \langle m_t, m \rangle \notin RI)$
3. $\forall m_i \in MI \backslash \{m_t\} \exists m_j \in MI : \langle m_i, m_j \rangle \in RI$
4. $\forall m_i \in MI \forall in \in m(IN) \exists_1 m_j \in MI : \langle m_j, m_i(in) \rangle \in RI$

The conditions two to four are similar to a pipes & filters approach. They express that there is a terminal model in MI whose output is O, that any output of a non-terminal model is the input for some other model, and that each input for a model is provided by only one other model. Wang and Liu show that the completeness of a combination model relation is equivalent to these four criteria. Within the proof, they present an algorithm which builds a composite model for a complete combination model set.

Despite these constructive parts, the work of Wang and Liu belongs to the more theoretical contributions in the area of model interoperability, as they explore *options* for connecting models, but do not express any way for concrete implementation. This, of course, is easier if a superstructure is available, which any models can interface to.

The approach of McBrien and Poulovassilis (1999) is a good example for work in this direction. As explained already in section 2.3.2, they propose a transformative technique for hypergraph structures. Based on these inter-model transformations, they introduce the concept of *inter-model edges*, linking structures between objects that belong to different modeling languages. They characterize this technique as

> *"[...] particularly powerful when a data model contains semi-structured data which we wish to view and associate with data in a structured data model. For example, we may want to associate a URL held as an attribute in a UML model, with the web page resource in the WWW model that the URL references."* (page 345)

McBrien and Poulovassilis show an example for a mixed model that results when integrating an ER (Entity Relationship) model or a web pages model with a UML representation. Figure 2.11, taken from this example, illustrates the linking of two models with an inter-model edge, table 2.3 contains the underlying representation in the hypergraph data model. An interesting aspect is that the hypergraph data structures used by McBrien and Poulovassilis are expressive enough to allow for the specification of models as well as inter-model constructs. The limits of their approach are, however, in the lack of expressing both dynamic aspects, and semantics beyond constraint checking.

Structured Modeling

In contrast to the majority of interoperability approaches which essentially focus on building some kind of explicit connection between different modeling languages, the technique of structured modeling (developed in the 1980s) is inherently interoperable to some degree, since its design is very open and it separates definitional concepts

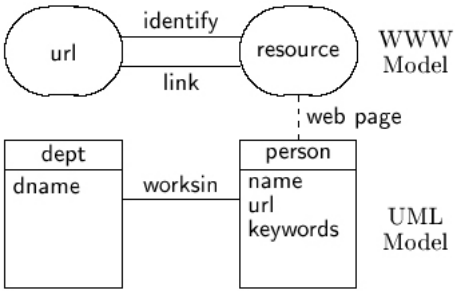

Figure 2.11: The association of models via inter-model edges

Table 2.3: Representation of an inter-model edge in the HDM Scheme $\langle r, c, a\rangle$

Edge	$\langle \_, www : r, uml : a\rangle$
Links	$\langle www : r\rangle, \langle uml : a\rangle$
Cons	$\langle \_, www : r, uml : a\rangle = \{\langle r, a\rangle \mid \exists z, s, us, pw, h, pt, up.$ $\langle\langle s, us, pw, h, pt, up\rangle, r\rangle \in$ $\langle www$:identity, www :url, www :resource$\rangle \wedge$ $a = s\circ$':'$//$'$\circ us\circ$':'$\circ pw\circ$'@'$\circ h\circ$':'$\circ pt\circ$'$/$'$\circ up\wedge$ $\langle z, a\rangle \in \langle \_, uml : c, uml : a\rangle\}$

from instantiations and algorithmic parts very clearly. Structured modeling aims at providing a systematic way of thinking about models and their implementations, and it is intended to serve as a foundation for generic computer based modeling environments (Geoffrion, 1989a).

The principal approach of structured modeling is based on the idea that every model consists of a collection of attributed entities which can be related to each other in several ways. One basic aim within structured modeling is to offer a flexible definitional system for entities and their relations that allows the use of structured modeling in various contexts. The underlying principles of the structured modeling definitional system according to Geoffrion (1989a) are the following:

Correlation. Definitions of entities in structured modeling allow references to other entity definitions. Thus, correlation of things can be expressed easily.

Acyclicity. The definitional interdependencies of elements in structured modeling are always *acyclic*, which facilitates algorithmic aspects dramatically.

Classification. Structured modeling provides five basic classes (see below), every defined entity belongs to exactly one of these classes.

Grouping. Similar definitions can be partitioned into groups, which allows the reduction of task complexity for the user.

Hierarchy. Groups of definitions can be organized hierarchically, which adds structure to models and further reduces complexity.

The elementary classes of structured modeling that implement these principles are:

Primitive Entities. These are the smallest distinctly identifiable entities.

```
NUTRi /pe/
MIN (NUTRi) /a/ : Real+
MATERIALm /pe/
UCOST (MATERIALm) /a/
ANALYSIS (NUTRi,MATERIALm) /a/ : Real+
Q(MATERIALm)/va/ : Real+
NLEVEL (ANALYSISi,Q) /f/ ; @SUMm(ANALYSISim*Qm)
T:NLEVEL (NLEVELi,MINi) /t/ ;NLEVELi>=MINi
TOTCOST (UCOST,Q) /f/ : @SUMm(UCOSTm*Qm)
```

Figure 2.12: Example of a model definition in SML

Compound Entities. Tuples of other (primitive or compound) entities.

Attributes. Tuples of entity elements together with a value in a specific range. A value can also be explicitly *not set*, indicating an incomplete model.

Functions. Similar to attributes, functions are tuples of entity elements together with a value in a specific range. Function values can depend on other involved valued elements, the association of values is done through rules.

Tests. Test elements are similar to function elements, but just allow for a boolean result value.

By means of functions and tests, structured models can be computationally active in the sense that mathematical expressions can be embedded and dynamically applied. However, it is worth noting that structured modeling only foresees *abstract* rules - the provision of an implementation for specific rule expressions is left open. This is done on purpose to avoid the restriction of the application area.

Well-defined entity sets without definitional gaps (i.e., references to entities that do not belong to the set) are called *closed elemental structures*. On top of these, *generic structures* and *modular structures* are defined. These serve partitioning and ordering purposes and thus contribute to the grouping and hierarchy characteristics. They complete the definition of a model instance in the sense of Geoffrion (1989a).

Model schemes are defined as specific sets of model instances which satisfy certain isomorphism characteristics with respect to elements, partitions and orderings. A trivial case of model instances that belong to a common model scheme results from attribute value variation.

The definition of model schemes can be done in SML (Structured Modeling Language). Here, the formal definition of a genus (which essentially is a group of definitions) includes the name of the genus, a statement about the dependencies, a type statement, the data type definition for attributes, and a mathematical expression for functions or tests. Figure 2.12 taken from Geoffrion (1987) illustrates an SML definition for a feedmix model. The primitive entities are lists of nutrients (`NUTRi`) and materials (`MATERIALm`), attributes are the minimum daily requirements per nutrient (`MIN`), the unit cost per material (`UCOST`), an analysis of nutrient-material combinations (`ANALYSIS`), and a quantity of chosen material (`Q`). The functions included in the model give the nutrition level (`NLEVEL`) and the total costs (`TOTCOST`), and a test determines whether the chosen material combination is sufficient (`T:NLEVEL`).

Using this framework of structured modeling and SML as a definition language, Geoffrion (1989b) analyzes different techniques of combining models or even modeling languages. Based on a hierarchy that includes model instances, classes, paradigms and traditions, he lists ten possible types of model integration. In his

analysis, he claims that a significant portion of these ten types are either not very meaningful (e.g., a join of two model instances that belong to different model classes), or unrealistic, like the integration of whole modeling traditions. As a consequence, he focuses on joining different model classes, and joining model instances that belong to the same class.

Within this scope, Geoffrion describes both definitional and procedural integration strategies. Yet, although he is able to present a method for definitional integration for SML models, he states that a complete automation seems out of reach: in the general case, both syntax and semantics of the integrated model seem to be problematic. For the procedural integration of structured models, Dolk and Kottemann (1993) discuss the roles of the different associated solver components and propose an elementary model interconnection language that controls the solvers.

2.3.4 Discussion

Most of the approaches presented in this section contain valuable parts that are worth considering in the implementations within this thesis. In particular, the descriptive metamodeling frameworks as presented in subsection 2.3.1 cover an area that is of interest within this thesis: the provision of a superstructure that is able to describe and contain heterogeneous modeling languages.

Advantages of the MOF framework include its clear categorization which is, in addition, well suited for an object oriented implementation, and the central consideration of model repositories throughout the approach. Problematic points in the context of MOF are that it aims at a very general level and consequently cannot handle details of model semantics. This is not surprising, since even the UML (which is only one of the languages covered by MOF) is criticized for not providing any sharp model semantics definition (Harel & Rumpe, 2004). This lack of preciseness is a serious drawback which restricts the direct usability of MOF for the implementations within this thesis.

The technique of Domain-Specific Modeling is very appealing, since it allows for defining custom modeling representations, and promises the generation of highly usable program code without confronting end users with unnecessarily complicated intermediate (UML) notations. Used in the methodology promoted by Tolvanen and Kelly (2004), DSM seems to be a good and powerful mechanism that effectively meets business needs. Problematic issues are that model interoperability is not addressed at all, and seems to be difficult to achieve with the element-wise manual code translation technique. Also, the process of interactive model construction, which is an important factor in this thesis, is not really in the focus of the model-code mapping of DSM. Finally, the heterogeneity criteria for the targeted modeling framework in this thesis are not conform with the core DSM assumption that one domain per client is sufficient, and that therefore the manual coding steps are unproblematic because they are not frequently needed.

Reviewing MOF and DSM in an integrated manner, a joint consequence is that the *concepts* that both approaches make use of to describe models are well worth considering. Both encourage the use of object oriented principles for meta modeling. The ways of handling model semantics and operational concerns are either too general (in the case of MOF), or too specific to be easily usable for non-programmers (DSM). A suitable approach for the aims of this thesis will have to be somewhere in between these two extremes.

The transformative techniques as outlined in subsection 2.3.2 provide a formal means to deal with model structures that change over time, and thus have possible usages for describing modeling processes. In addition, transformation approaches can allow for switching between different semantically interlinked representations. This is likely to be be an essential feature for some of the modeling applications

that are targeted within this thesis. Thus, a compatibility with the transformation techniques is desirable. However, while important system functions could be facilitated by the use of a transformation based engine, the system *core* cannot: the basic philosophy of these techniques is automatic matching and transformation of model structures. This does not match the target of supporting interactive modeling processes in which arbitrary heterogeneous structures - even "wrong" ones which would usually not be the result of some well defined transformation step - are principally acceptable. The work of Lara and Vangheluwe (2004) is a good example for this argument. The flexibility of their mapping technique is extraordinary, but it is unclear in how far (even partially) "ill" structures could be simulated in their approach.

The presented work of McBrien and Poulovassilis (1999) offers a good formalization of model transformations, and a reasonable encapsulation of graph structures with constraints that is worth considering in the implementations within this thesis. They also offer a flexible approach for connecting heterogeneous models via joining edge elements and thus reach a certain level of semantic model interoperability. Drawbacks of their structures are, however, that dynamic elements and simulation functionality are not supported. In addition, the approach of reaching interoperability through using inter-model edges is inherently limited: neither *direct* connections of elements from different modeling languages, nor connecting *elements* (rather than edges) are supported. Of course, the enormous flexibility of the underlying hypergraph data structure can compensate for some of these limits.

Besides these transformation oriented techniques, also the work in the area of formal model interoperability as presented in subsection 2.3.3 is of importance within this thesis. As already discussed in the introduction, the classifications and analysis done by Dolk and Kottemann (1993) are of value as "criteria lists". Dolk and Kottemann (1993) are also among the few authors who identify necessary components of integrated modelings systems and link these to object oriented implementations. However, they are quite focused on databases and structured modeling, and do not offer any implementation that meets their criteria.

The model integration work presented by Wang and Liu (2003) is very interesting as they generically cover all models which accept input parameters and deliver output results. The major restriction of their approach is that they deliberately treat models as black boxes, which disallows for fine granular insight into models. This, however, is among the requirements for the implementations within this thesis, as constructive processes dealing with partial models and model connections are targeted to be supported. Furthermore, Wang and Liu (2003) do not provide any mechanisms *how* to interconnect models that could theoretically be integrated.

The technique of structured modeling (Geoffrion, 1989a) shows possible ways for model interoperability. Similar to the work of McBrien and Poulovassilis (1999), structured modeling attempts to find suitable representations for interrelated domain concepts and their semantics. One add-on of structured modeling consists of the fact that the approach also allows for including external algorithms (solvers) that operate on the data structures. The SML notation does not meet current standards for interchangeable data formats (e.g., XML), but it is a good example of specifying semantically enriched model definitions in a way that is not hardwired to a concrete modeling environment.

Structured modeling, however, has three inherent disadvantages:

- It allows only for acyclic relationships between entities. This obviously simplifies algorithmic treatment, especially on the side of the solvers, but drastically reduces the set of supported entity structures from graphs to trees.
- The notion of a structured model includes algorithmic aspects that work on models, but only in an abstract sense. Neither is there a general approach for

interpreting or simulating models, nor does a generic solver framework exist (Wang & Zeevat, 1998). Both could significantly facilitate the implementation of (even interoperable) solvers.

- SML does not support dynamic rules, a technique that might be necessary for certain realistic simulations.

Some of these limitations of structured modeling have been overcome in extensions of structured modeling. The work of Lenard (1993), e.g., includes the notions of actions and transactions and therefore allows for simulating certain discrete processes. However, none of the currently existing modifications to structured modeling retains or even extends the interoperability of the original approach and at the same time avoids significant parts of its limitations.

2.4 Summary and Conclusions

Subsections 2.1.3, 2.2.3 and 2.3.4 already discussed the theory presented in this chapter. This final section of the chapter shortly reviews these discussions with respect to relevance of specific theoretical fields or contributions to the aims of this thesis.

The most important aspects of graph theory lie in the formal roots and basic terminology that this field offers: the notions of graphs, nodes, and edges will directly be built upon in the definition of visual typed graphs is section 4.1. Higher order results of graph theory (like, e.g., algorithms or complexity findings) will only indirectly have an impact on the framework level that this thesis aims at - however, they might be important within particular modeling languages, e.g., for calculating model semantics.

Visual language theory has two very important contribution areas that will be reused: the formalization of visual structures (in particular graph based ones) being composed of objects and attributes that represent visual layout information, and the notion of constraints as contained in a number of theoretical approaches. These can serve well the need of expressing syntactical rules within specific graph based modeling languages (cf. section 4.2). In contrast to these areas, the aspect of language parsing (word membership problem) typically solved through grammars or logic based formalisms in visual language theory only has a limited direct applicability in this thesis: in an interactive modeling system, the user typically takes the role of a "production system", and particularly educational applications will also need to tolerate "incorrect" models created by the users. Of course, a higher-order model checking mechanism which, e.g., verifies a model against a known set of correct solutions, could make use of graph grammars or other process formalisms from the field of visual language theory.

Although the descriptive meta modeling approaches presented in this chapter are very heterogeneous, the object oriented approach adopted in all of them is a common point that is relevant for the framework of collaboratively usable modeling languages to be presented in chapter 4. The existing theory in the field of model interoperability (and integrated interpretation) shows two things that are relevant for this thesis: first, there is currently no established uniform solution or standard for model interoperability (even restricted to graph based structures) which would have to be adhered to. Second, there are indeed a number of single approaches that attempt to achieve interoperability in heterogeneous models. These concepts will serve as references for the interpretation of visual typed graphs in (cf. sections 4.6 and 4.7). Similar to the graph grammars discussed in the context of visual language theory, the transformation oriented meta modeling approaches only have an indirect

applicability in this thesis, as the direct focus is not on automatic model processing, but on an interactive system.

Chapter 3

A Review of Graph Based Modeling Tools

The previous chapter discussed the currently existing *theoretical approaches* that are relevant for the aims of this thesis. This chapter supplements this by reviewing and comparing the state-of-art *systems and applications* which are closely related to the targeted collaborative and interoperable modeling framework.

3.1 Criteria

This section discusses the criteria used for the selection of systems to be *included* in the comparison (requirements), and the criteria used *within* the comparison.

3.1.1 System Requirements

This thesis proposes a method and, built upon this, an implementation for collaborative modeling with graph based representations, emphasizing on interoperability and the support of multiple heterogeneous visual languages. Such a narrow focus is of course not reasonable as a requirements list for a system comparison - as section 3.3 shows, there are currently no systems which exactly fulfill these criteria. On the other hand, too unfocused requirements (e.g., the consideration of all graph based modeling tools, or all interoperable collaborative tools) would lead to a review of a very large amount of systems which are only loosely related to the core aspects of this thesis.

Consequently, the choice of required criteria is an important design decision. Aiming at a reasonable amount and focus of the tools to be included in the systematic review, I propose the following four criteria as necessary requirements:

Modeling tools. This criterion is related to the intended *purpose* of the tool: with respect to the focus of this thesis, only tools designed primarily for *modeling* are included in the comparison. However, the context of modeling (e.g., educational vs. general-purpose modeling tools) is not used as a separate criterion for selecting the tools for the review.

Graph based representations. As this thesis deals with graph based representations (cf. section 1.4), only tools which essentially rely on this kind of representation are considered in the review.

Multiple representations. Apart from the representation *type*, also the flexible support of *multiple* (graph based) representations is an explicit target of this

thesis. For this reason, only tools which offer a certain kind of flexibility in terms of representations they offer are included in the survey.

Interactive usage. Finally, the focus of this thesis is to allow for an interactive usage of the modeling environment. Thus, to restrict the number of tools included in the detailed review, only those which explicitly foresee an interactive usage mode are included.

These criteria are largely independent of each other (though the criterion of graph based representations can of course be seen in close connection to the multiple representations) and serve well the purpose of filtering out systems which are only loosely related to the scope of this thesis. Prominent examples of systems or system types which fulfil three of the four criteria (and are therefore worth mentioning, though not described in detail) are mentioned briefly in the following.

Interactive systems which rely (at least to a large extent) on multiple and graph based representations but do not focus on modeling include most visual programming languages, the AgentSheets system (Repenning, 1994) designed for programming and simulating multi-agent systems, the SimQuest environment (Joolingen, King, & Jong, 1997), which is oriented towards simulations rather than modeling, and graph based hypermedia suites like, e.g., XCHIPS (Wang et al., 2000).

Examples for flexible and interactive modeling tools that do not emphasize on graph based representations include modeling frameworks like Modelica (Tiller, 2001) or Brahms (Sierhuis, 2001). These two applications essentially rely on a largely textual representation format and are very closely related to programming languages, the latter oriented towards agent programming. Further examples in this category are the Science Learning Space presented by Koedinger, Suthers, and Forbus (1999) and the microworlds in the E-Slate system (Kynigos, 2002): both approaches address well tool interoperability issues, but do not focus on graph representations exclusively. E.g., in the case of E-Slate, the microworlds have graphical representations, including some with graph based ones, but this is not in the core of the developments.

There is a wide number of interactive modeling tools which rely on graph based representations, but do not offer multiple representational notations. This includes all environments that allow the direct manipulation of graph based modeling languages like, e.g., Petri Nets, Entity Relationship diagrams, or System Dynamics. There are even a number of *collaborative* applications in this field: here, examples include:

- COLER (Constantino-Gonzáles & Suthers, 2001), a collaborative Entity-Relationship editor,
- the CLE environment (Lauer, Ueberall, Horvath, Matthes, & Drobnik, 2003) with its shared conceptual graphs,
- the C-Chene energy chain modeling graphs (Baker & Lund, 1997), and
- even some special tools which offer graph based collaborative representations for handheld devices (Luchini, Quintana, & Soloway, 2003).

Finally, modeling tools that support multiple graph based representations but are *not* designed primarily for an interactive usage can be found in the field of visual languages (cf. section 2.2). Here, typical implementations are the Visual Language Compiler-Compiler VLCC (Costagliola et al., 2002) or the AToM^3 metamodeling framework. Both tools rely on graph grammars and focus on the automated transformation of graph structures rather than on interactive editing and manipulation (cf. discussion in subsection 2.2.3). However, it is worth noting that AToM^3 even

allows for running simulations (i.e., model executions in a narrower sense) based on graph grammar specifications.

Collaboration support is not included in the list of requirements, although it is a central aim of this thesis. Apart from the aspect that collaboration support is typically not independent of the criterion of interactive usage, there are only few tools which offer collaboration support features in addition to the other required criteria (cf. section 3.3). Thus, taking collaboration support as a required criterion would have reduced the list of systematically compared tools dramatically.

3.1.2 Comparison Criteria

With respect to the aims of this thesis, the review of the currently existing systems and technologies for collaborative and interactive modeling with heterogeneous graph based representations is guided by the following four criteria:

Extensibility

This first criterion deals with the question in how far the compared systems are not only capable of handling multiple representations (which is a requirement), but in how far additional representational notations can be externally defined and used within the system. If this is possible at all, the following three aspects play a role for determining the degree to which the criterion is fulfilled:

- Which technologies are used for the extension mechanism?
- How easy and well documented is the mechanism?
- Are the custom system extensions functionally limited in some sense (compared to "built-in" ones)?

Interoperability

This second criterion is closely related to the interoperability issues raised in section 1.5. In particular, the following two questions are used to determine the degree of interoperability:

Syntax. Can elements from different representational notations be "mixed", resulting in heterogeneous graph based models? Are there any mechanisms (e.g., constraints) to control the types of structures that the users can build?

Semantics. If the system offers an internal semantic representation of the graph based models, in how far is this semantics retained also for heterogeneous "mixed" models?

A complete fulfilment of this criteria is given in the case of syntactic and semantic interoperability being supported by the tool, including control mechanisms which allow the specification of syntax rules and semantic mappings across representational notations.

Operational semantics / Model simulation

The third criterion is related to modeling with formal languages that can be simulated or "run". Here, the following three questions determine the degree to which the criterion is fulfilled:

- Are active structures and model simulations *possible* in the system?

- How *flexible* is the system concerning simulations: are different algorithms (or simulation types) supported?
- Are the simulation algorithms externalized and *encapsulated* in a way which allows them to be reused? How are the technical interfaces that enable this functionality structured?

Collaboration support

The fourth criterion in the comparison is related to the collaboration support that the systems offer. Here, two different aspects will be considered:

Synchronous level. Are there mechanisms which allow the synchronous co-creation of models? Are these directed towards co-located or remote scenarios? How fine-granular (or: customizable) are the synchronous collaboration support mechanisms, and what are their technical foundations?

Asynchronous level. Is asynchronous collaborative work supported, e.g. through tailored archives, retrieval functions, collaboration partner recommendation services, or version management and notification services?

Of course, both collaboration support levels can also be addressed in a limited manner through tool independent technologies, e.g. using NetMeeting (NetMeeting Resource Kit, n.d.) and BSCW (BSCW Homepage, n.d.) systems. Yet, the tool review will only rely on tool-provided mechanisms that go beyond these generic approaches.

3.2 Graph Based Modeling Tools

The following subsections of this section contain (in alphabetical order) the tool reviews. Each subsection briefly describes relevant points concerning history, availability and purpose of the corresponding system, and characterizes it in terms of the comparison criteria listed before. Where appropriate, technical details of the tools are given.

3.2.1 Belvedere

Originally designed to help support problem-based collaborative learning scenarios, the Belvedere system is targeted towards the support of collaborative argumentation and inquiry processes. The system has been under continuous development for more than a decade (Suthers, Weiner, Connelly, & Paolucci, 1995), and currently serves, e.g., as a means for analyzing representational affordances in collaborative contexts. A version of Belvedere is available at Sourceforge.net.

The focus of Belvedere is not exclusively on graph based representations. In fact, Belvedere supports other formats (like, e.g., tables, or threads). Thus, the required criteria of *graph based* and *multiple* representations are only fulfilled in the sense that the environment *does* allow for several representations, including graph based ones, but is not primarily designed for different graph based representations used synchronously. In addition, the criterion of *modeling* as primary system purpose is not completely met, considering the aim of Belvedere as a tool for collaborative inquiry. Yet, as Belvedere allows constructing and reflecting on diagrams of one's ideas, using representations such as evidence maps and concept maps, the thematic proximity to qualitative modeling becomes clear. As Belvedere is one of the few available systems that offer interoperability and collaboration support, the

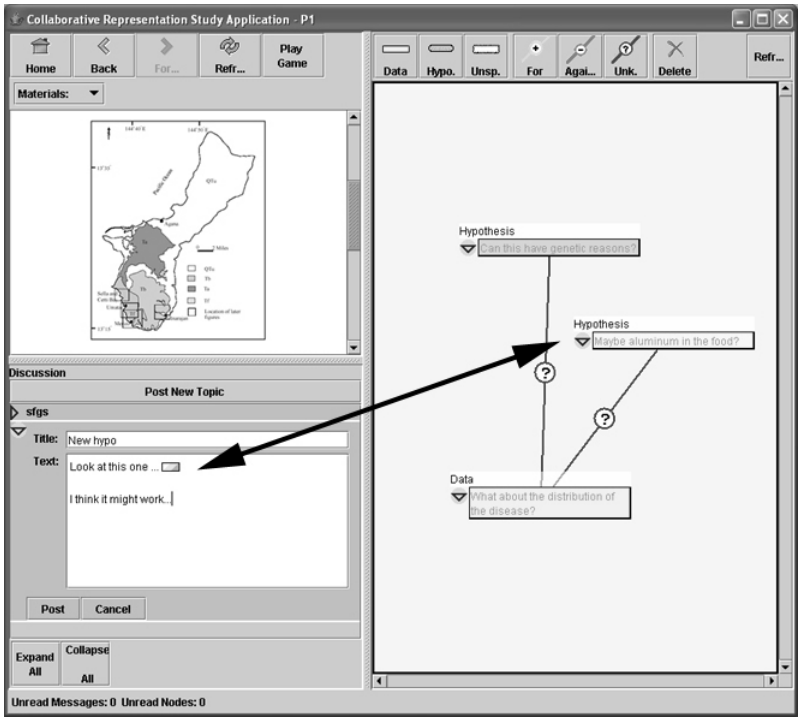

Figure 3.1: References between threaded discussions and graph structures in Belvedere

inclusion in the review (despite the fact that the required criteria are only partially met) is justified.

Extensibility is not in the focus of the BELVEDERE system at first sight: the tool does not offer to the user an inclusion of externally defined representational notations. However, current BELVEDERE versions (Suthers & Dwyer, 2004) can be parameterized with XML files that contain specifications of the representational primitives. The flexible software architecture builds good base for system extensions. Yet, the extension functionality is currently hidden and not externalized in a way that would allow "ordinary" users to realize and make use of the extension functionality.

BELVEDERE offers *interoperability* in the sense that references which connect different representations (e.g., threaded discussion and graph) are possible. These references consider the semantic types of the linked entities. However, there is no syntax constraint mechanism to restrict the set of allowed structures. Figure 3.1 illustrates this with a recent Belvedere version used in a study about collaborative representations: the forum posting (lower left corner) refers to an element in the graph (highlighted hypothesis).

Being designed towards argumentation and inquiry support rather than formal modeling, BELVEDERE does not offer any *simulation* or model execution functions. *Collaboration* support, however, is a strong point of the tool: from the very early versions on, BELVEDERE came with mechanisms to synchronize the representations created by the collaborators. The current version comes with a very flexible underlying communication API, based on nested message handler components and operational transformation techniques (Sun & Ellis, 1989) to ensure consistency. Due to the latter and the use of a dedicated communication server, current BELVEDERE versions are capable of supporting also asynchronous usage scenarios, in which edit

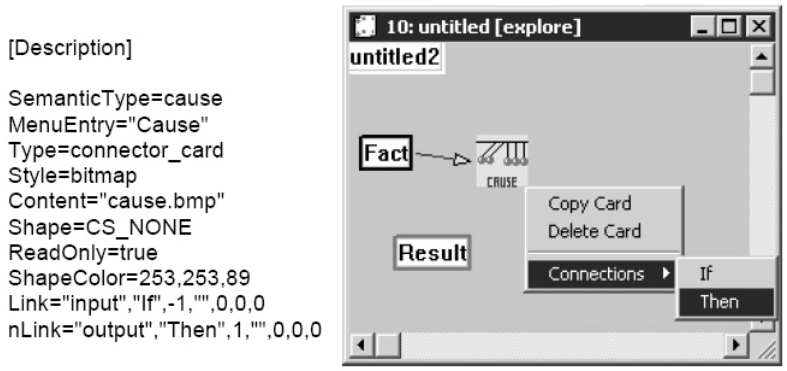

Figure 3.2: External definition of a relation primitive int the Cardboard system

events are transmitted with controllable delays.

Based on the Belvedere system (or integrating it), a number of specifically targeted applications have been developed in the past years. These include the COLER system (Constantino-Gonzáles & Suthers, 2001), which makes use of the Belvedere architecture and offers a tutor for the construction of Entity relationship diagrams, and the Science Learning Space (Koedinger et al., 1999), which integrates:

- simulation components in which experiments are run and data is collected,
- Belvedere as a representation construction tools in which data is analyzed and conceptual models are expressed and evaluated, and
- tutor agents that provide assistance.

In the Science Learning Space, these system components communicate via the MOO communication protocol.

3.2.2 Cardboard

Similar to Belvedere, the Cardboard system (Mühlenbrock, Tewissen, & Hoppe, 1997; Hoppe, Gaßner, Mühlenbrock, & Tewissen, 2000), developed at the University of Duisburg, is not specifically designed for modeling tasks, but aims more general at the support of collaborative problem solving and learning tasks which involve the use of shared external representations. The Cardboard tool is not maintained any more: in a sense, the Cool Modes environment presented in this thesis (cf. chapter 7) can be considered the successor of the Cardboard.

The Cardboard environment has a framework character in that it is generically *extensible* with visual language specifications. These specifications can be made externally in text files, which contain parameters of so-called content and connector cards. Figure 3.2, taken from Gaßner (2003), shows an example definition of a connector card together with the effect of this definition in the user interface.

A weak point of Cardboard is *interoperability* between different visual languages: upon creation, workspaces have to be parameterized with a specific visual language. Mixtures of elements from different languages are not supported.

The Cardboard system has been used in the areas of quantitative and qualitative modeling (Gaßner, Tewissen, Mühlenbrock, Loesch, & Hoppe, 1998). Here, semantic enrichment and certain operational features have been added to the environment through the external Prolog based DALIS component (Mühlenbrock et al.,

1997). Examples include domain knowledge bases for terms in algebraic relations. Though the DALIS interfaces were used for analysis and intelligent support functions primarily, they are theoretically also suitable for the implementation of (logic programming based) *simulation* or "model execution" functionalities. Apart from this, the CARDBOARD system does not include generic *internal* interfaces for model simulation.

Concerning *collaboration* support, the CARDBOARD application offers private and shared workspaces to the users, the latter enabled through an early C++ version of the MATCHMAKER library (Tewissen, Baloian, Hoppe, & Reimberg, 2000), which relies on the distribution of user interface events to coupled applications. Asynchronous usage scenarios or finer granular synchronization functions are not offered by the Cardboard tool. Based on the CARDBOARD/DALIS/MATCHMAKER tool combination, a number of further applications in the area of collaboration support have been implemented. This includes the collaboration analysis work of Mühlenbrock (2001), and the work of Gaßner (2003), who has used the CARDBOARD to support phases in group discussions by means of graph transformations.

3.2.3 Co-Lab

The CO-LAB environment (Skarmeta, Joolingen, Martinez, Celdrán, & Mora, 2002) is the software outcome of the European research project "Co-Lab: Collaborative Laboratories for Europe" (Co-Lab project homepage, n.d.). This project, which lasted from 2001 to 2004, focused on supporting collaborative inquiry learning in natural sciences at the upper secondary school level and the first years in university. To reach these aims, the CO-LAB environment provides students with a range of integrated experimentation options (including simulations and remote experiments with web cams), data visualization tools, communication facilities, and modeling tools. Technically, the CO-LAB environment relies on Java. It is designed as a completely web based system which does not require any specific software installations on the client machines.

Due to the proximity of the Co-Lab project to the aims within this thesis, an inclusion of the CO-LAB environment in this review makes sense, even though the system does not exclusively rely on graph based representations, and offers only one graph based modeling language (System Dynamics) - *extensions* of the CO-LAB environment with other modeling tools are not foreseen.

Similar to BELVEDERE, the CO-LAB system is highly *interoperable* in the sense that references between the different representations (tables, experiments, models, etc.) are generically supported. Due to the quantitative nature of the models that the CO-LAB system offers, syntactic integrity constraints are necessary and available.

Given the aims of the Co-Lab project, it is obvious that *simulation* and model execution are central points in the CO-LAB system. On a general level, this criterion is therefore completely fulfilled - especially taking into account the degree of interactivity that the system offers. However, concerning the specific criterion of simulating graph based models, the CO-LAB environment is restricted in that it exclusively focuses on the System Dynamics approach, not offering any alternative algorithms or modeling techniques.

One dedicated aim of the Co-Lab project is *collaboration support*. Therefore, it is not surprising that the environment has a number of strengths in this respect: The CO-LAB system offers various options that allow students to work together and share information. In terms of synchronous collaboration support, CO-LAB provides (among other features) shared workspaces to collaborate while creating and simulating System Dynamics models. Here, a strict floor control mechanism is employed, granting active model manipulation rights only to one user at a time.

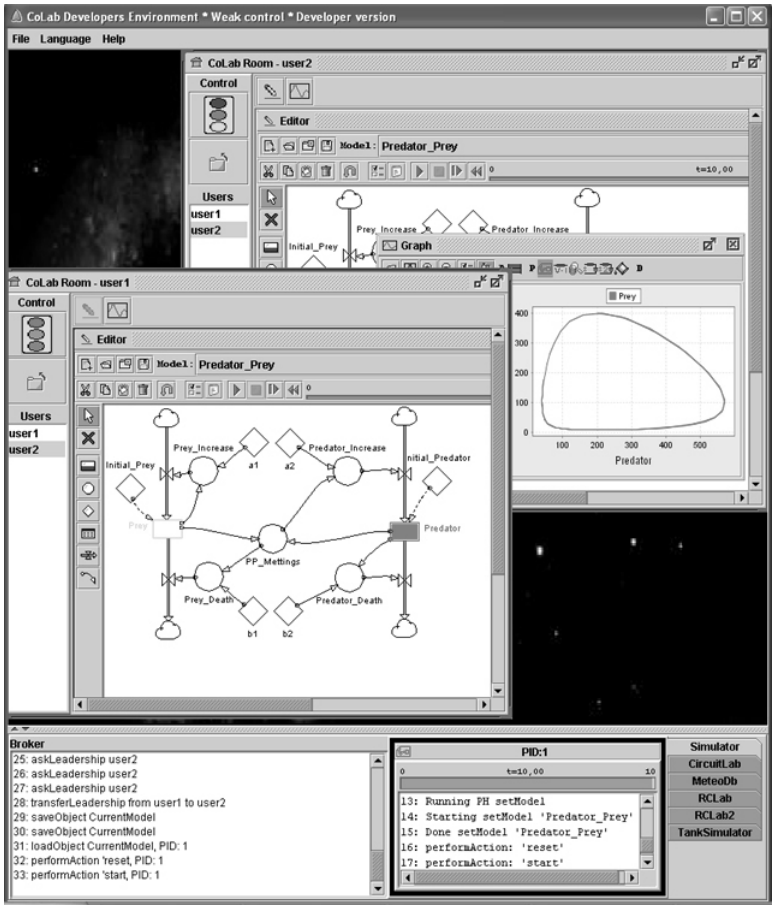

Figure 3.3: Synchronized predator-prey models in the Co-Lab environment

This situation is illustrated in figure 3.3. Here, two shared models are visible, with a traffic light indicating the floor. On the level of asynchronous collaboration support, the Co-Lab environment offers broker services to bring together interested learners, and searchable archives that contain the results of user's work.

3.2.4 Daidalos

The Daidalos system (Tsintsifas, 2002) has been created within a PhD thesis. The system is a part of the DATsys (Diagrammatic Assessment Teaching System) environment, which is designed for diagram based computer based assessment. Currently, Daidalos is not publicly available.

Daidalos is the meta-diagram editor of the framework, and can be used to model graphically diagram elements and their relationships as well as associated domain data models. It even allows the specification of tools to manipulate the diagram elements. As such, the criterion of *extensibility* is fulfilled: Daidalos can be seen as a visual editor for modeling languages, embedded in the DATsys framework. A wide variety of diagram types has been created with Daidalos, including e.g. Entity Relationship diagrams, Mind Maps, Petri Nets, Chemical Diagrams, and Analog Circuit Diagrams (DATsys Homepage, n.d.).

There is no explicit notion of syntax constraints in the Daidalos tool. However, the DATsys framework allows (by means of the embedded Ariadne component)

the task-dependent specification of primitives that the user has access to. Therefore, the criterion of syntactic *interoperability* is (at least minimally) met. As DATSYS and DAIDALOS address diagrams rather than models, there is neither an explicit concept of semantics, nor any means of model *simulation*. Yet, a distinctive aspect of DAIDALOS as embedded in DATSYS is the integration with the ARIADNE component, allowing for exercise specification and automatic assessment based on marking algorithms for the user-constructed diagrams. These algorithms can be seen as externally defined specifically targeted simulations. In addition, the DAIDALOS software architecture is suitable for including external simulation algorithms through used Command patterns (Gamma, Helm, Johnson, & Vlissides, 1995) - however, in DAIDALOS this pattern is used only for *visual* diagram changes, as no explicit notion of semantics is available.

Collaboration support is not addressed by DAIDALOS directly. However, the DATSYS framework contains various archives with corresponding retrieval and submission functions (all designed towards the targeted use in educational scenarios), so that a basic support for asynchronous co-operation is available, even though the focus of the environment is not set on this.

3.2.5 Dome

The DOME (Domain Modeling Environment) system (DOME specification, 1999) is an extensible collection of integrated model editing, metamodeling, and analysis tools following the domain-specific modeling method (cf. subsection 2.3.1). DOME is distributed by Honeywell and freely available (DOME Homepage, n.d.). Technically, DOME is based on Visual Works, and thus does not rely on current state-of-art technologies.

Similar to the other DSM-oriented tools in this review (GME and METAEDIT+), DOME is designed primarily for model-based software development. However, DOME differs essentially from these tools in two aspects: first, DOME comes with a built-in scripting language (Alter) and a visual data flow oriented programming language (Projector). These two languages allow the creation of program code translations for models (a crucial point in the DSM method) smoothly embedded into DOME. Second, DOME is not exclusively designed for synthesizing executable software from models, but also allows for other ways of "running" models.

Similar to the other DSM implementations, DOME fulfills the *extensibility* criterion. It comes with the embedded ProtoDOME tool, which allows the visual specification of modeling languages, including visual parameters of the representation as well as element attributes and their associations to Alter/Projector code. Language definitions can be stored externally in text files (using a proprietary format), which include the interpretable programming code and the specification parameters of the notations.

DOME offers a very limited notion of syntax constraints which essentially relies on the multiplicity of associations (cf. figure 3.4). *Interoperability* between languages is addressed in a restricted way: there is a "Shelf" component, which allows the reuse of single element primitives in other contexts. Furthermore, models can be composed of sub-models, i.e. hierarchical relations between models are allowed. The type of two models in a hierarchical relation does not have to be the same. However, DOME does not allow for modeling language interoperability in a broader sense: mixed representations, consisting of elements coming from different languages, are not generically supported, and there is no explicit notion of modeling language interoperability. Of course, it can be argued that the integrated programming languages are capable of serving interoperability purposes - however, there is no guiding framework for this embedded in DOME.

As already stated, a strong point of DOME is its flexibility in terms of *simulation*

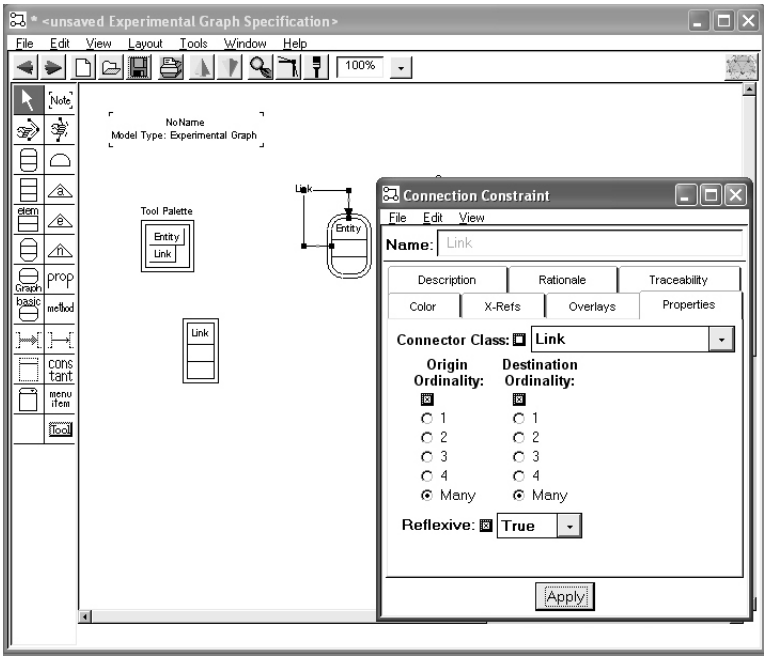

Figure 3.4: Language specification and connection constraints in DOME

support. The environment is (despite its foundations in DSM) not restricted to code generation, but allows for a wide variety of model executions - the built-in ones include, e.g., the simulation of Petri Nets by firing transitions. In contrast to MetaEdit+ and GME, DOME models can be interactively simulated, integrating the model construction with its simulation. This flexibility is reached through the interpretation mode of the embedded programming languages Alter and Projector.

A weak point of DOME concerning the criteria used for this system comparison is *collaboration* support. Here, no functionality is offered.

3.2.6 GME

The "Generic Modeling Environment" GME (Ledeczi et al., 2001) is a freely available metamodeling environment developed at Vanderbilt University. It is loosely related to the domain-specific modeling technique (cf. subsection 2.3.1), and is primarily a toolkit for model based program code synthesis.

In difference to MetaEdit+ and Dome (cf. subsections 3.2.5 and 3.2.7), GME adopts a modified DSM approach in the sense that a fixed number of high-level abstract and generic primitives (models, atoms, sets, references, and connections) is provided, from which models can be constructed (cf. figure 3.5). This is done in order to address the cost efficiency and reusability problems of DSM - it can be argued that the approach adopted by GME offers an intermediate level between UML and DSM in the sense that code generation is generically supported, a certain flexibility concerning representational primitives is provided, and re-use of models is partially possible.

As GME is primarily a metamodeling environment, the criterion of *extensibility* is met almost by definition: the primary goals of GME are the specification of modeling languages and the specification of translations for models into executable program code. However, one could argue that through the comparatively prescriptive categorization of primitives (as opposed to other metamodeling frameworks),

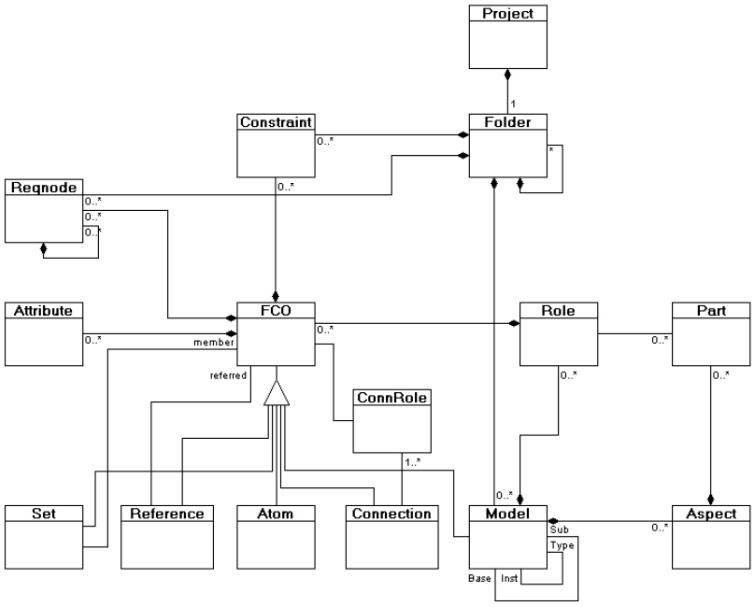

Figure 3.5: Modeling concepts in the GME environment

expressiveness and flexibility of the system are slightly restricted. These primitives and their relation are shown in figure 3.5 taken from Ledeczi et al. (2001). The figure illustrates the comparatively large amount of generic concepts, compared to "classical" DSM tools.

GME offers constraints to specify the set of syntactically correct models. As these constraints may also (by means of OCL) relate to attribute values, *interoperability* is addressed also concerning model semantics. However, GME does not contain any functions specifically directed towards the support of mixed heterogeneous models. Of course, on the program code level, these translations can be defined - yet, GME does not provide any libraries for this task.

As already stated, GME allows the synthesis of program code based on models constructed by the users. This can be seen as the model *simulation* functionality offered within GME. As GME contains interfaces to a variety of programming languages (e.g., C++, Visual Basic, and Java), a certain flexibility is available here. Yet, other ways of model simulation (apart from code generation) are not supported, which leads to an only partial fulfilment of the simulation criterion.

Finally, GME does not contain any built-in features designed to support *collaboration*.

3.2.7 MetaEdit+

MetaEdit+ is a metamodeling tool which adopts the domain-specific modeling approach (cf. subsection 2.3.1). It is commercially distributed by the Finnish MetaCase company (Metacase, n.d.). The inclusion of MetaEdit+ in this comparison was a borderline case, as the system is not primarily focused on graph based representations: domain-specific modeling in general and MetaEdit+ in particular also allow for other representational choices. Yet, MetaEdit+ relies on objects and their relationships primarily, and uses a graph representation as one of the built-in notations.

In analogy to the other metamodeling environments in this survey, MetaEdit+

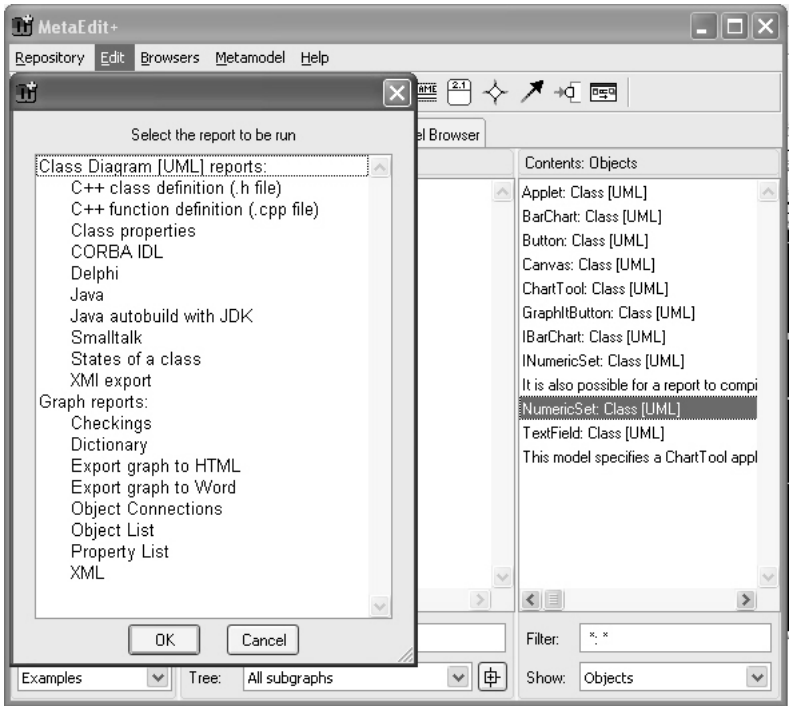

Figure 3.6: Examples for model execution options in MetaEdit+

fulfills the *extensibility* criterion - the DSM approach even *relies* on the user specifying "his" modeling language.

A strong point of MetaEdit+ is its public API, which allows an easy access of the tool from other applications via Web Services. Furthermore, MetaEdit+ completely relies on XML representations for models, and includes database connectivity. On a general level, all these functions contribute to *interoperability*. Yet, in a more special sense of syntactic and semantic interoperability as discussed in subsection 3.1.2, MetaEdit+ lacks certain features: the system does provide the option of defining syntactic constraints (rules which restrict the connection of certain element types). Yet, semantic interoperability between elements of different languages is difficult to achieve in the domain-specific modeling approach, as this essentially relies on tightly interrelated and "known" components with fixed code transformation rules. Here, the integration of external "unknown" components (which might have code translations that emerged from different contexts) is problematic.

Similar to GME (cf. previous subsection), MetaEdit+ allows the transformation of models into code, which can be considered as one type of model execution or *simulation*. Due to the DSM approach, the programming language to which the models are mapped is quite flexible (as the code translation for each primitive has to be explicitly specified only once). This is illustrated in figure 3.6, which shows possible code translations and other model executions in MetaEdit+.

MetaEdit+ is one of the few existing metamodeling tools which offer some kind of *collaboration* support: the core part of the tool distribution includes a shareable object repository with notification services, sub-model relations, and access right management, yet without commenting or version management. As such, the criterion of asynchronous collaboration support is partially met (yet, MetaEdit+ does not offer any functions for synchronous collaboration support).

3.2.8 ModelIt

The ModelIt system (ModelIt Homepage, n.d.), distributed by GoKnow (a demo version is freely available), is a visual modeling and simulation tool which is designed specifically for educational usage scenarios. ModelIt has been in use and continuous development for more than a decade (Soloway et al., 1997), current versions of the system are written in Java.

The main aim of ModelIt is to support students in building, testing, and evaluating models without requiring them to know the underlying calculus driving these models. Yet, ModelIt has only one fixed built-in calculus (simulation based on difference equations). As such, the decision to include ModelIt in this review is a critical case due to the only partial fulfilment of the criterion of multiple representations. The decision to include the tool in the review is based on the fact that, though ModelIt only offers one set of abstract primitives, the visualization of these is very flexible, leading to a variety of visual representations which have internal data models of the same type.

This restriction of expressiveness has an impact on the *extensibility* of the system: ModelIt allows the specification of new concepts and relationships visually through dialogs. Yet, the common frame set by the fixed calculus cannot be exceeded. Therefore (compared to other tools in this survey), the extension quality of ModelIt is only on a basic level.

A positive aspect of the common data models for primitives and generic calculus in ModelIt is that "mixed" models are unproblematic. It is possible to interconnect arbitrary components - ModelIt is always able to conduct a simulation of the resulting structure. Thus, the level of *interoperability* is very high, only reduced by the aspect that ModelIt does not foresee any kind of control elements to restrict the set of syntactically correct models.

As already stated, ModelIt supports simulation of models using the built-in calculus which cannot be exchanged. This lack of flexibility leads to an only basic fulfilment of the *simulation* criterion. Furthermore, ModelIt is not designed towards *collaborative* usage.

3.2.9 ModellingSpace

The ModellingSpace software (Avouris, Margaritis, Komis, Saez, & Melendez, 2003; Margaritis, Fidas, Avouris, & Komis, 2003), a successor of the ModelsCreator tool, is an outcome of the European research project "ModellingSpace" (ModellingSpace project homepage, n.d.) which lasted from 2001 to 2004. Similar to Co-Lab, ModellingSpace aims at supporting students in collaborative modeling tasks. Differences to Co-Lab are the focus of ModellingSpace on modeling in integrated real-time and asynchronous collaboration scenarios, and on the analysis of collaborative modeling processes. The Java based ModellingSpace software is freely available. However, it is not further developed any more. Its successor Synergo (BSCW Homepage, n.d.) does not allow for active models in the sense that a simulation can be run - instead, Synergo emphasizes on collaboration support.

ModellingSpace models use graph representations, the latter consisting of typed objects (called entities) and associations (called relations). The entity concept is kept very flexible in the software architecture: there is an abstract entity superclass which encapsulates name, description and visual representation of the entity, and is capable of handling an arbitrary number of typed attributes. Entities can have states based on attribute values, and the visual representation (i.e., the image) of the entity may depend on its state (Avouris et al., 2003; Avouris, 2004). ModellingSpace has a built-in editor that allows a user to easily define entities (cf. figure 3.7) and thus *extend* the system with custom entity libraries. A

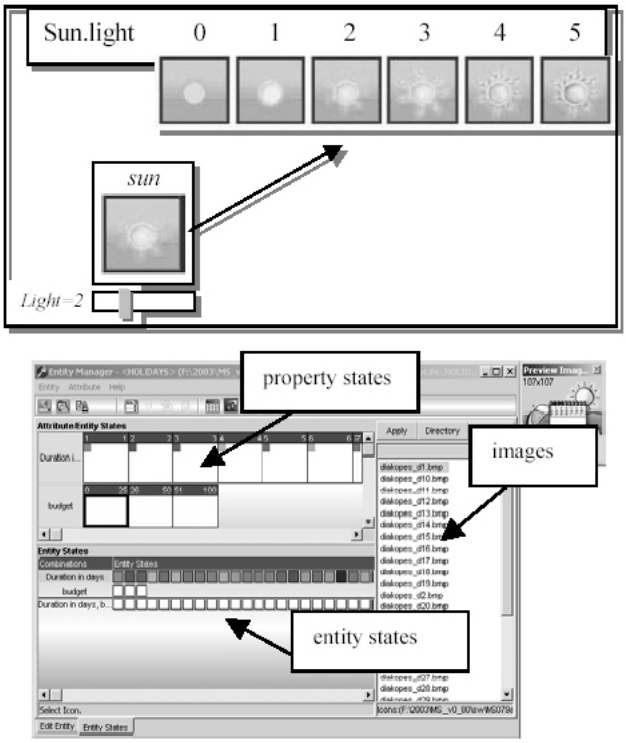

Figure 3.7: The entity editor of ModellingSpace

similar mechanism for relations is missing, though. The fixed set of relation categories (qualitative, semi-quantitative, and quantitative) cannot be extended, and new relation types cannot be added to the categories.

Interoperability is a strength of ModellingSpace: the system has a built-in mechanism for transferring entity definitions from one application to another one during the collaboration process (Margaritis et al., 2003). Neither in the entity editor nor on the software architecture level, ModellingSpace foresees advanced mechanisms to restrict the set of syntactically correct models. Only the built-in quantitative relations have some implicit conditions attached: e.g., it is not possible to relate qualitative entities to each other using quantitative relations, and certain self references are prevented (cf. figure 3.8). Using this small set of conditions, the ModellingSpace system ensures that only the set of structures that can be created with the primitives are valid models.

ModellingSpace has a built-in simulation function which is enabled if there is at least one semi-quantitative entity in the model. Similar to ModelIt, the simulation engine is fixed: only a simulation based on differential equations is supported. Therefore, despite the smooth integration of the simulation functionality into the system, the criterion of *simulation* is only minimally fulfilled.

Collaboration, however, is very well supported in ModellingSpace. The system relies on a replicated peer-to-peer architecture with lightweight model sharing mechanisms for synchronous collaboration (Margaritis et al., 2003). The users can choose between floor control and an uncontrolled mode, and also relay servers to facilitate synchronous collaboration are available. In addition to the peer-to-peer functionality, ModellingSpace offers model repository servers and special "community servers". These dedicated applications are used for awareness purposes, asynchronous message exchange, group and session management, and as a means

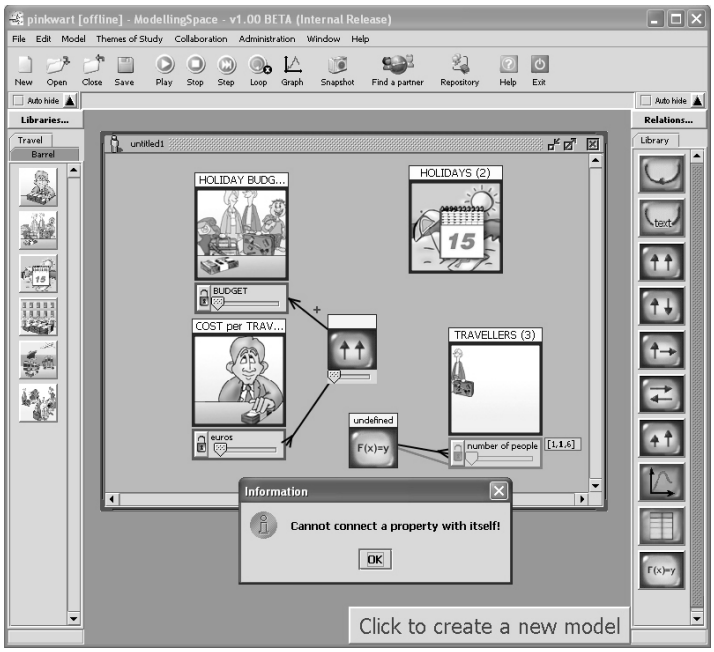

Figure 3.8: The ModellingSpace interface with a feedback message about a relation condition

that allows users to find collaboration partners.

3.2.10 MULTIGRAPH

Compared to most of the other tools in this survey, the MULTIGRAPH architecture and system implementation (Sztipanovits et al., 1995) is a relatively old approach which is not maintained any more. MULTIGRAPH is designed for building complex embedded systems based on a model-integrated approach. The underlying models, which can be of heterogeneous types, use graphs as conceptual data structures.

The MULTIGRAPH architecture consists of several components, including graphical model builders, model databases, and model interpreters. Model views (used in the graphical model builders) and interpreters constitute domain modeling languages, which can be dynamically added to the architecture. However, Sztipanovits et al. (1995) admit that writing a model interpreter is a complicated task which requires in-depth knowledge of the system internals. Therefore, the MULTIGRAPH approach fulfills the *extensibility* criterion only partially.

MULTIGRAPH allows for hierarchic composition of models: models can contain sub-models of the same or different types. In addition, models can be interconnected through explicitly defined "module interfaces" which control outward visibility of models. By means of these two functions, the MULTIGRAPH system reaches a certain degree of *interoperability* between models of different types. The architecture also allows for an integration of model interpretation results. Yet, MULTIGRAPH does not foresee a real *mixture* of elements from different modeling languages: heterogeneous model graphs consisting of primitives from different languages are not generically allowed.

With its multiple model interpreters, the MULTIGRAPH system is suitable for "executing" models flexibly. Yet, the purpose of the system is exclusively on code synthesis (in the context of embedded systems) based on domain models: the

flexibility is given in terms of target operating systems. Similar to GME, other ways of model *simulation* are not available in MULTIGRAPH. Therefore, the fulfilment of the corresponding criterion is only partial.

Although designed as a distributed system based on CORBA, the MULTIGRAPH architecture and system implementation do not address *collaborative* usage at all: the distribution serves system level interoperability purposes only.

3.2.11 Ptolemy

The PTOLEMY software (Hylands et al., 2003) is developed by an informal group of researchers at the University of Berkeley. The software has a history of more than 15 years, with early versions written in Lisp and C++. The current version, PTOLEMY II, is based on Java and available in open source form (Ptolemy project homepage, n.d.). PTOLEMY aims at supporting heterogeneous modeling, simulation, and the design of concurrent systems. Here, the focus is set on the generation of executable code based on model specifications - however, compared to other systems with similar target in this review (DOME, GME, and METAEDIT+), a primary objective in the design of PTOLEMY is to allow for interoperability. PTOLEMY does not exclusively rely on graph based representations - the system only operates on abstract components that do not have a visual representation attached. However, the visual editor VERGIL, which is delivered with PTOLEMY, adds graph structures as default representational notation to the PTOLEMY suite.

Models in PTOLEMY are composed of actors as primitive elements. Actors have associated ports, which define the incoming and outgoing interface of an actor. Ports can be connected with relations, which can be conceived as channels though which data flows. The well documented architectural design of PTOLEMY is very flexible concerning actors and their underlying concepts of data types and expressions: it is possible (and foreseen!) to *extend* PTOLEMY with customized actors. However, due to the complexity of the system this requires advanced programming skills and detailed information about some system internals: even in the technical system description, the "cardinal sin" of copying and pasting program code instead of using inheritance mechanisms is recommended for reasons of simplicity (Brooks et al., 2003, page 174). A metamodeling extension of PTOLEMY, which would facilitate the specification of actors, is planned.

PTOLEMY uses a rigid type system for the data that can be sent across ports, and a correspondingly strict interface specification for actors. Based on this, syntactic *interoperability* is reached: the system controls the fulfilment of these syntax conditions while the user is constructing models. In addition, PTOLEMY allows for a hierarchical composition of models. Figure 3.9, taken from (Hylands et al., 2003), shows how PTOLEMY models are composed of the mentioned primitives.

Based on this syntactic interoperability, PTOLEMY allows for lending semantics to models through different models of computation, which

> *"[...] form design patterns of component interaction, in the sense that Gamma, et. al. describe design patterns in object oriented languages."*
> (Hylands et al., 2003, page 8)

PTOLEMY comes with a number of built-in "domains", which encapsulate models of computation. These include, e.g., discrete event models, finite state machines, process networks, and continuous time models. These domains serve as a means to flexibly *simulate* or execute PTOLEMY models under different higher-level concurrency perspectives. In addition, actors are generally independent of domains. This adds a further degree of flexibility to the system, as models are domain polymorphic (i.e., they can be used in different interaction models). The inclusion of new domains (i.e., simulation engines) into PTOLEMY, is possible, but a complex task due

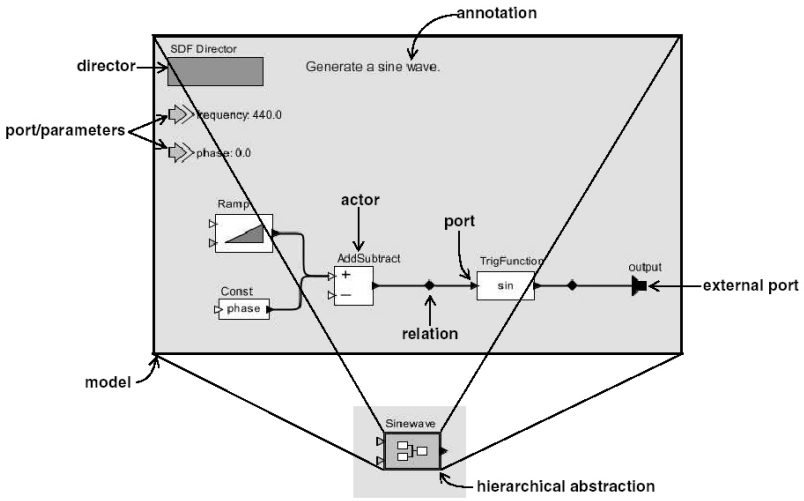

Figure 3.9: The basic model concepts in Ptolemy

to the complexity of the system. Similar to the case of the actors, a metamodeling add-on is planned here.

Although a rudimentary support for distributed usage of Ptolemy exists in the software (several interfaces for remote method invocations), Ptolemy currently does not foresee a *collaborative* usage.

3.2.12 Visio

Visio is a part of the well-known commercial Microsoft Office Suite (Microsoft Office, n.d.). The tool is designed for the creation of diagrams in business and technical contexts. In particular, Visio aims at facilitating the representation of ideas, processes, and systems. Visio fulfills all the requirements for inclusion in this review - yet, it does not exclusively focus on graph structures but also supports other forms of visual representations.

The primitives used to create diagrams in Visio are called "shapes", which are organized in libraries that represent "drawing types". Shapes can be defined in a visual editor that is very expressive in terms of defining the visual appearance of shapes - if Visio is used in the Microsoft Windows tablet edition, also a definition of shapes using handwriting is supported. Beyond the visual appearance of shapes, the developer version of Visio also allows for advanced shape specifications using, e.g., ActiveX and .NET technologies. Thus, Visio completely fulfills the *extensibility* criterion - however, advanced programming skills are required to handle the options available in the developer version.

Since shapes are the basic primitive of Visio, the system does not offer any encapsulation of visual languages as such: the drawing type libraries which contain the shape masters merely serve as container objects and have no functionality attached. As a consequence, Visio does not offer any kind of language *interoperability* apart from the fact that shapes can be arbitrarily moved between different containers. A basic form of syntactic interoperability, however, is available in the sense that shapes from different containers can be arbitrarily connected: apart from the options in the developer edition, Visio does not offer any simple generic mechanism for defining syntax rules.

Visio is designed primarily as a tool for creating diagrams. Therefore, the focus

of the system is set on visual elements, not on *simulation* functionality. Due to this, the tool does not provide any generic framework for "executing" diagrams. In the developer version, however, a lot of programming languages (including ActiveX, .NET, and Visual Basic for Applications) can be used, and current VISIO versions also have interfaces based on web services technologies. With all this, the construction of shapes or libraries of shapes with simulation functionality is indeed possible - yet, the support that VISIO as a framework adds to this is only marginal.

Like all the other applications in the Microsoft Office Suite, VISIO fully supports the NetMeeting technology (NetMeeting Resource Kit, n.d.) for sharing application views. In addition to this, VISIO also neatly integrates with the Microsoft SharePoint server technology (Microsoft Windows Server, 2004) which is designed to support *collaboration*. It provides shared document spaces with the option of editing and commenting on other's documents, generic communication tools (e.g., chats), and presence awareness functionality. Of course, the collaboration support that the SharePoint server integration adds to VISIO is not specifically designed for the needs of collaboratively used graph based representations.

3.3 Discussion

The previous section contained structured descriptions of the twelve modeling tools included in this review. Despite the relatively strict requirements list, the review contains a variety of differently targeted tools. Some (BELVEDERE, CARDBOARD, CO-LAB, DAIDALOS, MODELIT, and MODELLINGSPACE) have educational purposes, others like DOME, GME, METAEDIT+, MULTIGRAPH, PTOLEMY, and VISIO are targeted more towards different professional usage scenarios.

Though of course, an implicit usage of some basic theoretic concepts (e.g., from graph theory) can be found across all tools, the relation between the tools and technologies presented in this chapter and the theoretic approaches discussed in chapter 2 is surprisingly weak. Only the DSM approach (cf. subsection 2.3.1) is represented though three tools: DOME, GME, and METAEDIT+. There are no other theoretic approaches (e.g., from the fields of model interoperability or model transformation) with associated implementations that meet the requirements for inclusion in this review. For the case of visual language based tools, this has been discussed in section 3.1.1: here, the lack of interactive usage (which is usually the effect of the graph grammar engines employed in these tools) is the required criterion that is often not met.

As an attempt to systematically compare the tools along the criteria list presented in subsection 3.1.2, table 3.1 gives a summary of the modeling tool descriptions. The assessment symbols used in the table express the degree to which the tools fulfil the criteria. They have the following meanings:

+ Completely fulfilled
(+) Partially fulfilled
∘ Basically prepared but not elaborated
- Not fulfilled

The criteria used for the comparison are high-level and only partially operationalized, and the borders between the categories of fulfillment are not sharp. Though this implies a certain *interpretation* concerning the assignment of assessment symbols, it takes into account both the complexity of the criteria and the possible variations in their fulfillment. In addition, it leads to a much better focus of the comparison table.

Already a simple row-wise analysis of table 3.1 immediately reveals several things. First, there is no single application which fulfills all four criteria at least

Table 3.1: Comparison of graph based modeling tools

	Extensibility	**Interoperability**	**Operational Semantics**	**Collaboration Support**
Belvedere	(+)	(+)	-	sync: + async: ∘
Cardboard	+	-	∘	sync: (+) async: -
Co-Lab	-	+	∘	+
Daidalos	+	∘	∘	sync: - async: ∘
DOME	+	(+)	+	-
GME	+	(+)	(+)	-
MetaEdit+	+	∘	(+)	sync: - async: (+)
ModelIt	∘	(+)	∘	-
ModellingSpace	(+)	(+)	∘	+
MULTIGRAPH	(+)	(+)	(+)	-
Ptolemy	(+)	+	+	-
Visio	+	∘	∘	sync: ∘ async: (+)

partially ((+) symbol). The tools with the most positive assessments (weighing all criteria equally) are DOME, ModellingSpace, and Ptolemy. Interestingly, these three environments come from very different domains: DOME is an older metamodeling environment that integrates interpreted programming languages to make up for the disadvantages of the DSM method, ModellingSpace is an educationally oriented system with focus on collaboration support, and Ptolemy can be considered as an experimental system which is designed towards model interoperability. DOME and Ptolemy lack collaboration support features, whereas ModellingSpace has certain limitations concerning the representable structures and their simulation.

A first column-wise analysis yields that the criterion of extensibility is by far the one that is best fulfilled. This may be due to the decision of selecting only tools which operate on multiple representations for this review. Often, the step between *multiple* and *externally defined* representations is not too big. The criterion of interoperability is slightly better fulfilled than the simulation/operational semantics one, and the criterion of collaboration support (both synchronous and asynchronous) is by far the least met one. Only two environments (Co-Lab and ModellingSpace) have very good support for both synchronous and asynchronous scenarios. One observable pattern is the correlation of the system purpose to the fulfilment of the criterion: apart from ModelIt, all educationally oriented tools in the review offer some collaboration support functions. For the professionally oriented tools, the image is different: here, only Visio and MetaEdit+ offer some (limited) collaboration support functionality. A similar pattern can be observed for the criterion of operational semantics/simulation: none of the systems that gets at least a (+) assessment is an educational one. This may be due to the fact that an educational purpose is already a specific system purpose, which may lead to a lower degree of flexibility in model simulation compared to multipurpose tools.

An analysis of criterion pairs reveals that the criteria in this review are largely independent. There are only two exceptions to this: there is no non-extensible system which is capable of flexibly simulating models (this may be due to the fact that a high degree of flexibility is only *required* in extensible systems), and the criteria of collaboration support and operational semantics seem to be mutually exclusive: there is no single system in the review which fulfills both. As stated above,

this also correlates with the purposes of the systems (educational vs. professional).

Using a 3-out-of-4 columns analysis approach, some other characteristic application patterns become visible: in this review, there are no tools which meet all criteria but extensibility. This is largely due to the fact that almost all systems are extensible. MetaEdit+ is the only environment which has its only weak criterion in interoperability. The applications with good results in all categories except for simulation (Belvedere and ModellingSpace) can be characterized as flexible "shared representation" tools. These tools have strengths in other areas than simulation: e.g., Belvedere is capable of bridging between different representational forms (also different from graph based ones), and ModellingSpace has internal features for video editing, which allows for an easy specification of attribute dependent image representations for conceptual objects. Finally, the non-collaborative applications which meet all the other review criteria are professional systems which focus on model interoperability: DOME and GME as metamodeling tools, and the model-integrating systems MULTIGRAPH and Ptolemy.

Apart from these table analysis, two qualitative observations concerning the tools in the review can be made:

- Some educationally oriented tools (e.g., ModelIt and ModellingSpace) only allow for syntactically correct structures that can be simulated. This leads to a very easy usage of these tools, as virtually no errors can be made. Yet, this characteristic goes along with a lack of flexibility concerning simulation flexibility. Compared to this, other environments, primarily the DSM based tools and Ptolemy, are very flexible concerning simulation/model execution types - however, their use is often quite complicated and sometimes even requires programming skills.

- The degree of collaboration support greatly varies among the tools in the review. For the specific case of synchronous collaboration support, all the tools which offer this (Belvedere, Cardboard, Co-Lab, and ModellingSpace) rely on shared workspaces techniques. However, none of these systems focuses on finer granular synchronization support mechanisms below the whole model as shared primitive. The graph structure of the models and its potential for synchronization purposes is usually not further exploited, although the communication architectures of these systems would theoretically allow for this step.

3.4 Challenges

This chapter, and in particular the discussion in the previous section, shows that there is currently no system which fulfils all the four main criteria used for the review. Yet, as argued in the introduction, such a tool might be very helpful for various application areas, in particular in educational contexts. As such, the development of a suitable conceptual approach and a corresponding system implementation are challenges worth pursuing.

A particular outcome of the detail discussion in section 3.3 is that there is an open challenge in finding a way of integrating the following two aspects, both conceptually and technically:

- synchronous collaboration support for modeling tasks that involve rich and active structures that can be simulated and "run", and

- an extensible framework that allows for various kinds of interoperability between graph based modeling languages

The next chapters of this thesis show how, based on current technology, these challenges can be met, also taking into account the more fine-granular criteria already presented in section 1.5. Interoperable and synchronously usable graph based models are conceptually treated in chapter 4 - based on these results, a flexible system architecture which retains expressiveness and operational semantics (i.e., computational power) for the supported structures, is discussed in chapter 6. Finally, collaboration support functions and the extensible Cool Modes framework (as a reference implementation of the architecture) are shown in chapter 7.

Chapter 4

The Reference Frame Approach

The general aims for the implementations within this thesis have already been described in detail within the previous chapters, in particular within the sections 1.3 and 1.4. Very briefly summarized, these are the development of (1) a conceptual method to support collaborative modeling with graph representations, and (2) a flexible and interoperable framework which implements this method.

I consider it reasonable to split the presentation of my approach into three parts that incrementally base on each other and that vary in their degree of abstraction. The first part, presented in the sequel of this chapter, is on a *conceptual level* and contains basic definitions and formalisms, abstract methodological issues, and theoretical perspectives. The second part, described in chapter 6, contains an *abstract high-level implementation approach* and a discussion of architectural and algorithmic aspects related to the implementation of the developed conceptual framework. Finally, the third part presents an *example system implementation* that puts the abstractions into practice.

Taken together, these three parts illustrate the adopted methodology which consists of a conceptualization of models and modeling in graph representations that is consequently taken up and driven forward towards a flexible computational approach. An implementation of this approach by a framework that explicitly takes into account the specific needs of collaborative usage concludes the line of thought and development.

The criteria and concrete aims as worked out in previous chapters of this thesis, in particular in section 1.5, relate to different levels of abstraction and thus to different parts of the description. It makes, e.g., sense to handle usability issues on the level of a concrete application, whereas general flexibility and expressiveness concerns are better answered in more abstract parts. Therefore, the following chapters (except for chapter 5) integrate discussions about the fulfilment of the criteria at appropriate places in the text. In addition, each of the chapters concludes with some summary remarks about the achievements of the respective chapter with respect to the aims and challenges of this thesis.

4.1 Typed Graphs and Layouts

Among the basic concepts to deal with when aiming at the development of a methodology for collaborative modeling with graph representations are, of course, graphs with their nodes and edges (cf. section 2.1). The second important area is modeling, and in particular modeling language formalizations as discussed in section 2.3

- these typically make use of classification schemes to distinguish between different object types. This motivates the following first definitions:

Definition 4.1 *Let $\mathcal{N}$ be a set of elements called* node types*, and let N be a set of nodes. Then a mapping dom : $N \mapsto \mathcal{N}$ is called a* node type mapping*. The image of dom, written dom(N), is called* node domain *of N. Edge type mappings (i.e., mappings from edges to edge types) and edge domains (images of edge type mappings) are defined in analogy.*

For certain concepts developed in later parts of this thesis, it will be helpful to allow types to be specializations of other types. This is expressed in the following definition:

Definition 4.2 *Let $\mathcal{N}$ be a set of node types. Then a reflexive, antisymmetric and transitive relation $\sigma \subseteq \mathcal{N} \times \mathcal{N}$ is called* node subtype relation *on $\mathcal{N}$ if $(N_1, N_2) \in \sigma$ expresses that all nodes of type N_1 are also of type N_2. Edge subtype relations are defined in analogy as transitive relations between edge types.*

The concept of type mappings for nodes and edges models type relationships. If a type mapping for a set of nodes and edges is available, then this can be transferred to the graph structure level as follows.

Definition 4.3 *Let $\mathcal{N}$ be a set of node types, and let $\mathcal{E}$ be a set of edge types. Let G=(N,E) be a graph, and let $dom_N : N \mapsto \mathcal{N}$ and $dom_E : E \mapsto \mathcal{E}$ be node type and edge type mappings. Then G is called a* typed graph *over ($\mathcal{N}$,$\mathcal{E}$).*

Leaving out subtype relations (which can be understood as nodes or edges having several types at the same time), the type mappings obviously induce an equivalence relation on a typed graph, with the equivalence sets being defined by the types contained in $\mathcal{N}$ and $\mathcal{E}$. The notion of typed graphs is the syntactical definition that builds the foundation for all the advanced concepts and approaches to be presented in the sequel. Therefore, the following observation is important: typed graphs do not restrict syntactic interoperability. The following proposition shows that it is possible to mix structures of different types arbitrarily without leaving the general context of typed graphs - in a certain sense, a closeness property.

Proposition 4.1 *Given an index set I and (for each $i \in I$) node type sets $\mathcal{N}_i$ and edge type sets $\mathcal{E}_i$. If G=(N,E) is a graph with $N = \bigcup_{i \in I} N_i$ and $E = \bigcup_{i \in I} E_i$ so that for each $i \in I$ the following holds:*

- *$\mathcal{N}_i \supseteq dom(N_i)$ and*
- *$\mathcal{E}_i \supseteq dom(E_i)$,*

then G is a typed graph over ($\bigcup_{i \in I} \mathcal{N}_i$,$\bigcup_{i \in I} \mathcal{E}_i$).

Proof. We have to show that the node and edge domains of G are subsets of $\bigcup_{i \in I} \mathcal{N}_i$ resp. $\bigcup_{i \in I} \mathcal{E}_i$. This is easy to see: according to the prerequisites in the proposition, $\mathcal{N}_i \supseteq dom(N_i)$ for each $i \in I$, and therefore $\bigcup_{i \in I} \mathcal{N}_i \supseteq \bigcup_{i \in I} dom(N_i)$. On the other hand, we know that $\bigcup_{i \in I} dom(N_i) = dom(\bigcup_{i \in I} N_i) = dom(N)$. The proof for the edge domains is analogous.

The concept of typed graphs offers the syntactic base for further considerations and is a suitable means to represent abstract graph structures that consist of entities of different types. The following definitions introduce a means to consider visual representations of typed graphs. Within this thesis, this is of course an essential aspect, as the intended applications will deal with visual graph representations.

Definition 4.4 *Given two sets V_N and V_E, called* visual node attributes *and* visual edge attributes*, then a pair $L=(\lambda_N, \lambda_E)$ of mappings with $\lambda_N : N \mapsto V_N$ and $\lambda_E : E \mapsto V_E$ is called a* layout *of a Graph G=(N,E).*

Definition 4.5 *Given a typed graph G=(N,E) over $(\mathcal{N},\mathcal{E})$ and a layout L of G, then $\langle G, \mathcal{N}, \mathcal{E}, L\rangle$ is called a* visual typed graph *over $(\mathcal{N},\mathcal{E})$. In short, I will refer to G as a visual typed graph if the other parameters are unambiguous.*

This definition of a layout for a graph is abstract in the sense that it does not prescribe concrete sets V_N and V_E. This design choice was made in order to take into account the variety of different description frameworks for visual parameters as discussed in section 2.2. In particular, the openness of the layout definition allows connections to the logic based and algebraic techniques presented in that section. For some implementation parts within this thesis, however, concrete and explicit definitions of V_N and V_E will be necessary (cf. subsection 6.1.2).

In definition 4.4, the visual attributes of a graph are not defined in dependence of particular node or edge types. This would obviously be an alternative. The chosen approach was motivated primarily by two arguments:

1. In the concrete applications implemented within this thesis, the layout attributes are not mere theoretical constructs, but have practical usages for displaying graph representations. Here, mixed and potentially unrelated attributes may lead to significant rendering problems, unless transformation functions are given.

2. Heterogeneous and type dependent node and edge attributes are enabled by their semantics (cf. section 4.3). In this sense, the taken approach does not prevent the inclusion of type-dependent attributes that relate to the visual representation of elements.

Together with figure 4.1, the following example illustrates the concept of typed visual graphs.

Example 4.1 *A graph G=(N,E) with $N = \{n_1, n_2, n_3\}$ and $E = \{e_1, e_2\}$, with type sets $\mathcal{N} = \{circle, rectangle\}, \mathcal{E} = \{line\}$, and type mappings dom_N and dom_E constitutes a typed graph. Consider the following example definition for the mappings:*

$$dom_N(n_1) = circle, dom_N(n_2) = rectangle, dom_N(n_3) = rectangle$$

$$dom_E(e_1) = line, dom_E(e_2) = line$$

"Natural" visual attributes for planar representation of nodes are the lower left and upper right corner of the rectangle that outlines the node. Accordingly, let $V_N = \mathbb{R}^2 \times \mathbb{R}^2$. The edges are characterized by their starting and end point and a curvature measure, which is expressed in $V_E = \mathbb{R}^2 \times \mathbb{R}^2 \times \mathbb{R}$. The following example values for the layout mappings in $L=(\lambda_N, \lambda_E)$ complete the definition of $\langle G, \mathcal{N}, \mathcal{E}, L\rangle$ as a visual typed graph.

$$\begin{aligned}
\lambda_N(n_1) &= ((0,0),(10,10)) \\
\lambda_N(n_2) &= ((10,20),(30,40)) \\
\lambda_N(n_3) &= ((20,0),(30,10)) \\
\lambda_E(e_1) &= ((5,10),(10,30),0)) \\
\lambda_E(e_1) &= ((30,5),(30,30),3))
\end{aligned}$$

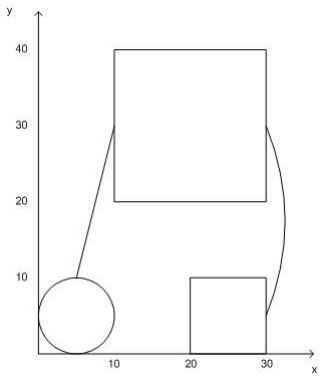

Figure 4.1: Representation of the visual typed graph of example 4.1

In addition to showing a concrete case of a typed visual graph that uses the definitions, example 4.1 reveals three characteristics of the chosen layout approach.

- Visual attributes have to be *interpreted* in order to obtain a visual representation: syntax and values of the layout mapping are not sufficient. The definitions do not contain any formal description of, e.g., a "circle" or the concept of "curvature". Thus, in order to generate concrete visual representations, an interpretation of the data is necessary.

- The example clearly illustrates that visual edge attributes may *depend* on the nodes that an edge connects, or on their visual attributes. In the particular example, the positions of the start and end point of an edge are obviously related to the corresponding node, though not completely determined by it. Despite being worth considering, this dependency between visual node and edge attributes is not problematic in general: on the theoretic level, edges are even defined on the base of nodes (cf. definition 2.1), and practical implementations will easily allow for interrelated visual attributes, as will be shown in the next chapters of this thesis.

- It is obvious that the relation between visual graph representations and visual typed graphs is not unambiguous. In the present case, alternative definitions of node type sets (e.g., only one very general type) and visual attributes (e.g., an attribute that determines a geometric shape of a node) can lead to identical visual representations. In general, the approach offers design choices between the pieces of visual information that are *indirectly* represented through types, and the ones that are made *explicit* by means of attributes. Given that for the targeted application area, a unique mapping between visual graph representations and visual typed graphs is not a necessity, I consider the mentioned ambiguity (that could also be denoted flexibility) of the conceptual base more a strength than a weakness of the approach.

4.2 Integrity Constraints

The previous section introduced the notion of typed graphs. As stated, this notion enables syntactic interoperability in the sense that it does not put any restrictions on the covered structures: any graph together with suitable domain mappings for nodes and edges is a typed graph.

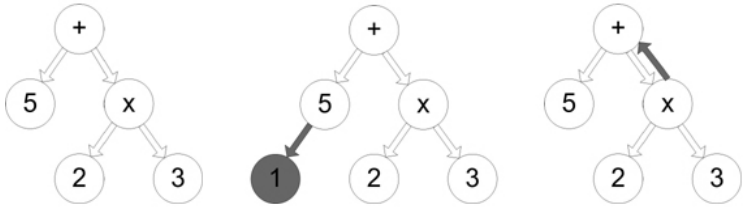

Figure 4.2: A syntactically correct calculation tree (left) and two incorrect graphs (center and right)

While the openness of this definition is a key for supporting mixed and heterogeneous models (cf. next sections of this chapter), it lacks one central feature: a lot of modeling languages define certain constraints that have to hold in order to call a graph based representation a valid model. These constraints are typically needed as a basic prerequisite in order to allow the computation of formal semantics, or (on a more application oriented level) conduct simulations of models. Two simple examples illustrate this:

Example 4.2 *Conceived as a graph based structure, a Petri Net (Petri, 1962) consists of two node types: places and transitions. There is a formal Petri Net semantics that defines the "firing" of transitions and thus enables the simulation of a Petri Net - however, all formal semantics relies on the prerequisite that neither places nor transitions must be connected to an element of the same type, i.e., a* bipartite *graph is required.*

Example 4.3 *Known from compiler theory, a simple calculation tree can be built with node types that represent arithmetic operations, and a "number" type. The left part of figure 4.2 shows a simple calculation tree with value 11. Obviously, the set of meaningful structures (in the sense that a value can be computed) is restricted by some constraints:*

- *A node of type "number" must not have outgoing edges, as there is no reasonable calculation rule for this (cf. center part of figure 4.2).*
- *A node of type "+" or "×" must have at least one outgoing edge (though this could be relaxed by the reasonable assumption of default value 1 resp. 0).*
- *A node of type "-" or "/" must have exactly two outgoing edges (Similar to the point above, this could be solved by specific default mechanisms).*
- *Cycles are not allowed, as they would lead to infinite calculation sequences (cf. right part of figure 4.2).*

The examples 4.2 and 4.3 pointed out that both the abstract graph structure (e.g., forbidden cycles) and the types of nodes and edges can serve as parameters within constraints. Yet, they were of a relatively simple type. Obviously, more complex kinds of constraints in specific graph based modeling languages are imaginable, so that on a syntactical level, a general definition of a constraint mapping makes sense:

Definition 4.6 *A* constraint mapping *is a predicate* c *which determines the syntactic correctness of a visual typed graph by assigning it a boolean value. In the case of* $c(\langle G, \mathcal{N}, \mathcal{E}, L\rangle)$ *evaluating to* `true`, c holds *and the visual typed graph G is called* conform with c.

The syntax predicates can be thought of as "markers" which identify "correct" visual typed graphs according to some constraint criterion. Note that definition 4.6 does neither specify a concrete representation for constraint mappings nor a concrete calculus to evaluate the constraint predicates. This is left to concrete implementations of the Reference Frame approach - one concrete implementation example based on rules is shown in chapter 6.

As motivated by the above examples, constraint mappings will typically relate to the abstract graph structure and eventually consider node and edge types. Definition 4.6 does, however, also allow for syntactic integrity conditions that take into account the layout of the graph, and thus its visual representation. This design choice was made in order to be able to cover the variety of visual formalisms (e.g. Randell et al. (1992), Ligozat (1998), or Wang and Zeevat (1998)) within the formalism of the presented approach.

The proposed definition of constraint mappings does not separate generic constraints (which are applicable to arbitrary graphs irrelevant of their type and layout) from ones that are specific for certain node or edge domains. An alternative approach would consist of including constraint definitions with domain definitions. This approach would be more explicit in relating node and edge types to their inherent constraints, but would not support the desired interoperability between expressions with mixed domains. Section 4.5 shows how this conflict can be solved.

There are two aspects that the given notion of constraint mappings does not address. First, it does not even try to consider the semantics of visual typed graphs (cf. section 4.3). This, indeed, is a very interesting field (cf. chapter 9), as it leads to deciding whether a certain expression is "correct" also from a higher, potentially task-oriented, level (Herrmann, Hoppe, & Pinkwart, 2003). Yet, these questions exceed the scope of this thesis, and in particular also the purpose of the constraint predicates - restricted to the syntactic correctness, there are efficient implementations (as will be shown in the following chapters), which is not the case for general semantics oriented checks. The second issue not covered with the proposed concept of constraint mappings is its implementation. A generic interoperable framework will have to guarantee that, given an arbitrary set of constraint mappings, all the predicates are fulfilled at any point in time. As will be shown later in this chapter, this is a potential source of problems.

4.3 Expression Semantics

The concepts elaborated in the previous parts of this chapter provide a framework for graph structures that consist of laid-out differently typed entities, restricted through constraint mappings. The emphasis was put on flexibility and expressiveness. One aspect not considered up to here is the *meaning* of constructs. In the context of computational modeling, this meaning of structures is of course an important issue (cf. section 1.4 for the general interrelations between meaning and modeling). Therefore, it makes sense to include a place for expressing the meaning of visual typed graphs in the proposed conceptual framework.

In computer science, and particularly in theory oriented fields like, e.g., compiler theory, the term usually associated with meaning of expressions is *semantics*. Harel and Rumpe (2004) have recently analyzed the variety of different usages of the term semantics in current literature. They have opposed to this the original formal roots of computational semantics. I decided to adopt their approach due to its clarity, computational accessibility, and generality:

Definition 4.7 *Let $\mathcal{G}$ be a set of visual typed graphs. Then a couple (D,Ip), consisting of a* semantic domain *D and a* semantic mapping *$Ip : \mathcal{G} \mapsto D$ is called a*

graph semantics *for $\mathcal{G}$. For a concrete visual typed graph $G \in \mathcal{G}$, the image $Ip(G)$ of the semantic mapping function is also called* semantics of G *in short, if ambiguities are impossible.*

Similar to the definition of constraint mappings, the notion of graph semantics is independent of node and edge domains. At first sight, this approach may appear unusual, as typically the semantics of graph based representations (if defined at all) are tightly bound to a modeling language and its primitives. Yet, the chosen approach of (at least partially) decoupling semantics from concrete node and edge domains is an essential factor for reaching semantic interoperability, as will be discussed in sections 4.5 and 4.6.

The abstract definition of graph semantics does not include any link to possible implementations. Obviously, both semantic domains and semantic mappings may be quite complex for expressive modeling languages. Examples which demonstrate this complexity are the formal semantics of State Chart diagrams (Harel & Naamad, 1996) with its mappings to program code, or the semantics of Entity Relationship diagrams (Chen, 1976) which maps to database structures.

Though being defined on a high level of abstraction and independent of concrete implementations, semantic mappings may have several structural properties that have a clear impact on the possible implementations. In the following, four of these properties are shortly described, and an example for a concrete semantic mapping is given.

Decomposability. In a number of cases, it is possible to define graph semantics based on the semantics of the single nodes and edges. In these cases, semantic domains and mappings for node and/or edge types are available, and $Ip(G)$ can conceived as a function defined over terms like $Ip(n)$ and $Ip(e)$, which represent the semantics of single nodes and edges.

Regional bounds and effects. In the case of decomposable semantics, where single nodes and edges have a specific semantics, the semantic mapping functions for nodes and edges will usually not be context-free in the sense that a particular node or edge of a graph is mapped to the same element of the semantic domain irrespective of its neighbors in the graph. For this reason, the corresponding semantic mappings will be denoted with $Ip(n_G)$ and $Ip(e_G)$ in the sequel of this thesis, with G being a visual typed graph, n a node of G and e an edge of G. In this context, an interesting characteristic property of a semantic mapping is whether the subgraph that has an impact on a node (or edge) semantics can be regionally bound. This property has an immediate impact on synchronization contexts (cf. next subsection). In some modeling languages, the immediate neighborhood of a node plays a key role for these regional bounds (cf. example 4.4).

Stepwise computational complexity. The notion of the semantic mapping for visual typed graphs does not include any notion of computational complexity. Of course, any implementation of a specific mapping will have to deal with potential problems in this field. Given the intended application of an interactive environment which allows users to add and remove nodes and edges, an important characteristic property is the delta of computational complexity, which can be formulated as: Knowing $Ip(G)$, how complex is the calculation of $Ip(G')$ with G' resulting from G by adding/removing one node or edge? For decomposable semantics, this complexity will usually be tightly related to the complexity of single node or edge semantics.

Levels of dependence and independence. The notion of visual typed graphs,

upon which the semantic mapping is defined, is quite rich. In particular, it includes the following information:

- the *layout* of the typed graph (spatial information),
- the underlying abstract graph (structural information),
- the contained node types (object type information), and
- the involved edge types (connection type information)

Typically, a concrete semantic mapping will not use all this information, but only selected parts. The spectrum of really needed types of information having an impact on $Ip(G)$ is an interesting structural property of a semantic mapping.

Example 4.4 shows a concrete semantic mapping for the case of calculation trees, which are known from compiler construction.

Example 4.4 *In a calculation tree (cf. figure 4.2), edges do not have a specific formal semantics (except from their role in defining child relations). The semantic domain D for all node types is* $\mathbb{R}$*. The semantic mapping for a node* n *is defined as follows, with* $C($n$)$ *denoting the set of children of* n, $c_1($n$)$ *and* $c_2($n$)$ *denoting the first and second child of* n *in the tree, and* val(n) *standing for the value of a number node:*

$$Ip(n) := \begin{cases} \text{val}(n) & \textit{if } n \textit{ is of type "number"} \\ \sum\limits_{m \in C(n)} Ip(m) & \textit{if } n \textit{ is of type "+"} \\ Ip(c_1(\text{n})) - Ip(c_2(\text{n})) & \textit{if } n \textit{ is of type "-"} \\ \prod\limits_{m \in C(n)} Ip(m) & \textit{if } n \textit{ is of type "}\times\textit{"} \\ Ip(c_1(\text{n}))/Ip(c_2(\text{n})) & \textit{if } n \textit{ is of type "/"} \end{cases}$$

The semantics of the corresponding visual typed graph G (i.e., the whole calculation tree) can be identified with the semantics of the root element of the tree:

$$Ip(G) := Ip(n), \textit{with } n \textit{ being the root node of } G$$

In terms of the structural properties outlined before, the calculation net example can be described as follows:

- The semantic mapping is decomposable and has a trivial edge semantics.
- The semantics of a node depends on the semantics of its children. Therefore, the subgraph that has an immediate impact on the semantics of a node is composed of this node's children. Of course this leads to a recursion, so that the subgraph that indirectly has an impact on the node semantics is the subtree whose root is that node.
- Adding an element to a calculation tree can in the worst case cause the semantics of nearly all others to change (this is exactly when the tree is a degenerated list, and the new node is added as a second child to the last "inner" node). The complexity of a single addition or multiplication can be assumed as constant, so that the (non-recursive) calculation of a local node semantics is linear in the number of nodes. Together, this results in a worst case of $O(N^2)$, with N the number of nodes in the visual typed graph.
- The semantic mapping depends on the abstract graph structure and the node types, but not on the visual layout and the edge types.

An aspect worth noting is that our notion of semantics does explicitly refer to *visual* typed graphs. As such, it is possible to include the layout of a typed graph, i.e. spatial information, in the calculation of graph semantics. Compared to a number of existing approaches like, e.g., suggested by McBrien and Poulovassilis (1999) or Lara and Vangheluwe (2004), which explicitly focus on the abstract graph structure, this significantly extends the scope of covered languages. In the terminology of Costagliola et al. (2002), the definition of graph semantics as proposed in this section is suitable for the class of *hybrid languages*. Applications that make extensive use of these layout considerations for graph semantics will be shown in subsection 8.1.2.

With the availability of formal semantics, visual typed graphs are significantly extended: as shown in example 4.4, the possibility of attaching interpretation functions to expressions is the key that allows for automatic calculations and simulations. This augments the concept of visual typed graphs towards formal representations of models which additionally take into account representational parameters. All the frameworks that aim at supporting formal modeling languages contain some mechanism similar to the graph semantics as defined above. Three examples which make this explicit are the solver components in SML (Geoffrion, 1989a), the extents that encapsulate semantic states of objects in the work of McBrien and Poulovassilis (1999), and the graph attributes in the work of Lara and Vangheluwe (2004).

The notion of formal graph semantics as introduced in this section is of course only applicable to a subset of the graph based representational languages that are targeted within this thesis. As stated in the introduction (section 1.4), I explicitly intend to support languages with lower degrees of formality, such as UML, causal feedback diagrams, QOC, or concept maps. In these languages, the definition of a general computation rule determining the formal semantics of an expression is often not possible, although the expressions may have a well-defined and human-understandable "meaning".

Two missing pieces in the area of graph semantics have not been dealt with in this section. The first one is the way in which semantic information is attached to typed visual graphs (i.e., how the values of the Ip functions are stored), the second one relates to more operational aspects: beyond the availability of semantics, a modeling and simulation framework will require triggering mechanisms which initiate the calculation processes and thus allow for making the graph structures really active and interactive. Chapter 6 proposes solutions to these questions.

4.4 Synchronization Requirements

One of the distinguishing factors between the present work and comparable approaches in the domains of metamodeling and visual languages is that I specifically intend to allow for collaborative usage scenarios with flexibly shared representations. Though typically the concrete support for these mechanisms will be done on the concrete implementation level, there are two factors that can be considered already on the level of the formal conceptual level.

One of these factors is that modeling activities often consist of several phases, which may potentially be repeating and have no predefined order (Löhner et al., 2003). Phases are associated with activities and usage modes: two typical phases are, e.g., the editing and modification of a model vs. its interactive simulation. In synchronous collaborative scenarios, problems may occur if the co-users do not agree on a single joint usage mode. This would of course not be a problem to be solved by the computer in face-to-face situations, but in distance ones it has to be considered. Thus, it may be reasonable to foresee an indication of the current usage mode together with expressions in visual modeling languages. The implementation of this idea will be shown in chapter 7.

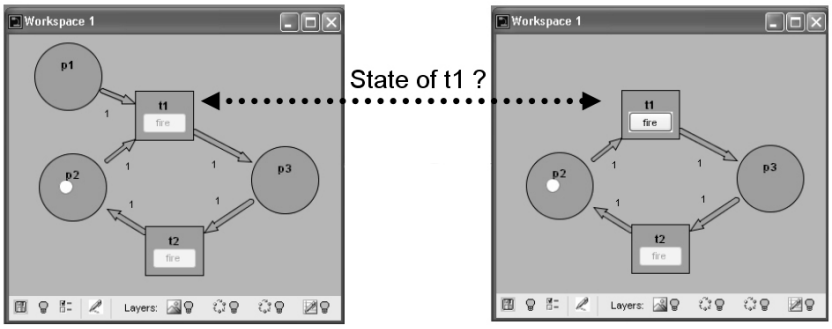

Figure 4.3: The problems of partial synchronization

Another important issue about shared graph structures is the possible discrepancy between flexibility of synchronization, and coherence or closure requirements of models. The logical consequence of aiming at maximum degree of flexibility in sharing graph structures is to allow for a synchronization of arbitrary substructures, i.e. subgraphs, to the extreme case of having only single nodes synchronized. These partially shared structures have interesting application areas and allow for flexible work modes. An example for this is the following: with partially synchronized graphs, it is possible for users to privately work on the construction of a model, and to publish only selected parts of it, e.g., a subgraph that contains some explicitly marked result elements. Insight into the way that these results were elaborated does not necessarily have to be granted. The private results can then be re-used in a variety of ways, e.g. to collect and compare results of different users, as demonstrated in the domain of mathematics by Kuhn, Hoppe, Lingnau, and Fendrich (2004), or even to build new models that rely on the results of the previous ones.

While this degree of flexibility may sound attractive, there are also situations in which partially shared models may be problematic. Apart from the general question how edges could be coupled without also sharing the nodes that this edge connects (this exceeds the notion of graphs and this is not dealt with in the present work), a critical point is that partially sharing models may lead to significant differences between the semantics of the shared model parts. This is due to the fact that the semantic mapping function is, in general, not context-free (cf. previous section). It is not only the general graph semantics that varies, but also that of single nodes which are contained in both of the partially shared models. Figure 4.3 illustrates this problem with the example domain of Petri Nets. The two workspaces are partially synchronized and differ only in the presence of one single place (p1) and its connection to the transition t1. This causes the semantics of t1 to change, and in particular also affects the semantics of the whole graph: the left net is dead, whereas the right one is non-terminating.

As argued, any general attempt to retain a common semantics between only partially shared (and therefore non-identical) models has to face the following problem: either one single shared semantics is preserved in the system and the result is a mismatch between local representation and system-internal semantics, or the semantics is only related to the respective local models. In the latter case, the problem is (as shown in figure 4.3) the non-existence of a common result.

One possible strategy to deal with this problem is to restrict the degree of flexibility concerning sharing entities. If the semantic mapping $Ip(n)$ of a node n does not depend on other entities, then it is reasonable to allow this node to be coupled independently of any other elements in the model graph. Otherwise, the (recursively determined) set of needed model elements has to be included in the set

of shared elements upon trying to share n:

Definition 4.8 *Let G=(N,E) be a visual typed graph with a semantics Ip(G), and let $n \in N$ be a node of G. If* n *has an associated semantic value (i.e., the semantic mapping $Ip(n_G)$ of* n *in* G *is defined), then a* synchronization context of n in G*, denoted by $Sync(n_G)$, is a subgraph of G containing n so that $Ip(n_G) = Ip(n_{Sync(n_G)})$. A function $S : N \mapsto \mathcal{P}(G)$ so that each node is mapped to a corresponding synchronization context is called* synchronization context mapping*. A function $\mathcal{S} : N \times G \mapsto \mathcal{P}(G)$ which accepts a node and a graph (containing that node) as input and returns a subgraph which is a synchronization context of the node in the graph is called a* generic synchronization context mapping*.*

Definition 4.9 *A synchronization context $Sync(n_G)$ is called minimal if no real subgraph of $Sync(n_G)$ fulfills the synchronization context condition for n in G.*

Proposition 4.2 *Let G=(N,E) be a visual typed graph with a semantics Ip(G), and let $n \in N$ be a node of G with defined semantic mapping $Ip(n_G)$. Then a minimal synchronization context of n in G exists but is, in general, not unique.*

Proof. A trivial synchronization context of n in G is obviously G itself, so that the existence is shown. The fact that $Sync(n_G)$ is in general not unique can be shown with a counterexample: a calculation tree consisting of the root node n_1 of type "×", and three child nodes n_2, n_3, n_4 of n_1 that are all of type "number" with $Ip(n_2) = Ip(n_3) = 0$ and $Ip(n_4) = 1$. Here, two different minimal synchronization contexts of n_1 in G are spanned by the node sets $N_1 = \{n_1, n_2\}$ and $N_2 = \{n_1, n_3\}$.

The proof of proposition 4.2 shows that the minimal synchronization context of a node in a graph can even depend on the values of semantic attributes. This means that upon a change in semantics (e.g., caused by a model simulation step), the minimal synchronization context may change. Using synchronization contexts as foundations for partially coupled models, this has to be taken into account: in collaborative work contexts, a non-minimal but stable synchronization context may be superior to a theoretically optimal but frequently changing one.

For a number of modeling languages, minimal synchronization contexts can be defined easily, as the following example illustrates for the case of Petri Nets (cf. figure 7.3 in the implementation part of this thesis for an XML representation that contains non-minimal synchronization context specifications):

Example 4.5 *Petri Nets are visual typed graphs that have the node type set $\mathcal{N} = \{place, transition\}$. For a visual typed graph G=(N,E), a minimal synchronization context mapping S is as follows (for reasons of simplicity, only the nodes that span the synchronization context graph are given):*

$$S(n) := \begin{cases} \{n\} & \text{if type(n) = "place"} \\ \{n\} \cup \{m \in G : (m,n) \in E \vee (n,m) \in E\} & \text{else} \end{cases}$$

This expresses that places can be synchronized node-wise, whereas the activation state and therefore the semantics of transitions depends on their input and output places, and thus on their complete neighborhood. Of course, the proposed synchronization context mapping (by definition) ensures consistent semantics only node-wise. In this example case, the activation state per synchronized transition and the tokens per synchronized place are guaranteed to be identical. However, the general model (i.e., Petri Net) semantics can still vary between partially shared models if only partial node sets are synchronized: e.g., one partially shared net can be dead, whereas the other is alive. Of course, another synchronization mapping (which synchronizes the whole connectivity component of the graph) can solve this problem.

A strict consideration of synchronization contexts solves the dilemma between coupling flexibility versus coherence of models. If minimal synchronization contexts are used, the solution can even be theoretically optimal in the sense that no "semantically unnecessary" elements are synchronized. Yet, even apart from the dynamics of the minimal synchronization contexts, one problem remains: there is no generic calculation algorithm for a minimal synchronization contexts. Especially in the case of modeling languages with non-formal semantics (like, e.g., concept maps), it is a priori unclear what such an algorithm should calculate. In chapter 6, some algorithmic approaches to solve at least the technical challenges will be discussed. Parts of the evaluation in chapter 8 describe usage situations that benefit from partial synchronization (involving synchronization contexts).

4.5 Reference Frames

The previous sections in this chapter have presented a number of ingredients needed for a methodology that supports collaborative modeling with graph based representations. In particular, I have identified a number of sets (e.g., node and edge types, or semantic domains), and a list of mappings (e.g., constraints, semantic mappings, or synchronization context mappings). All of them are important within a conceptual framework for collaborative modeling, and in some sense the elements are strongly related to each other: typically, all the concepts have to be combined in order to express the characteristics of a certain modeling language. E.g., a specific constraint mapping usually belongs to a particular set of node and edge types, and a graph semantics may in turn rely on syntactic integrity constraints.

For this reason, a central concept that bundles together all the ingredients makes sense. This can be conceived as the formalized abstraction of a visual modeling language itself, in contrast to the previous definitions which covered specific aspects of modeling languages. The fact that the central concept serves basically as a frame that allows the contained elements to reference each other motivates its name "reference frame":

Definition 4.10 *Let $\mathcal{N}$ denote a set of node types and $\mathcal{E}$ a set of edge types, and let V_N and V_E be visual node and edge attributes. For a set C of constraint mappings, a semantic domain D, a semantic mapping Ip, and a generic synchronization context mapping S, the tuple $\mathcal{R} = \langle \mathcal{N}, \mathcal{E}, V_N, V_E, C, D, Ip, \mathcal{S} \rangle$ is called a* Basic Reference Frame*. For a given Basic Reference Frame $\mathcal{R}$, $\mathcal{N}(\mathcal{R})$ is an abbreviated notation for the contained node type set, analogous notations will be used for the other concepts (edge types, constraint mappings, etc.).*

The definition of Basic Reference Frames as given above is open with respect to the interrelations between the contained elements $\mathcal{N}, \mathcal{E}, V_N, V_E, C, D, Ip$, and $\mathcal{S}$. This is done on purpose, both in order not to restrict the scope of Basic Reference Frames unnecessarily, and also because (due to the large scope of modeling languages covered by the concept of Basic Reference Frames) a generic formula that expresses *how* these elements relate to each other cannot be given in the degree of exactness and detail that is needed within a formal definition. However, some general level relations between the elements are the following:

- The constraint mapping set C operates on visual typed graphs and is defined over the general abstract graph structure, visual attributes, and the types in ($\mathcal{N}$,$\mathcal{E}$). C enforces the syntactic correctness of visual typed graphs with respect to the modeling language expressed by $\mathcal{R}$.
- The semantics of the modeling language is expressed by the couple (D,*Ip*). *Ip* maps the set of syntactically correct visual typed graphs (as specified by C)

to D. In the specification of Ip, both node and edge types (contained in $\mathcal{N}$ and $\mathcal{E}$) and visual attributes may serve as input parameters.

- Wherever syntax or semantics refer to the visual layout of typed graphs, they do so by means of the attributes contained in V_N and V_E.
- The generic synchronization context mapping $\mathcal{S}$ is defined for (at least) the node types of $\mathcal{N}$.

In this integrated manner, a Basic Reference Frame can serve as a means to enable co-construction (through the provision of element types, layout parameters, and synchronization context mappings) and interpretation (via syntax constraints and semantics) of expressions in visual modeling languages, as will be shown in chapters 6 and 7.

From a theoretical point of view, a Reference Frame can be conceived as the framework for a generating system for concrete visual typed graphs: the only lacking aspect is a component which instantiates graphs and handles the type mappings. This functionality could be gained through several techniques, e.g. by means of grammars. This shows the thematic proximity to approaches in the field of visual language theory (cf. section 2.2) - however, as this thesis is aimed towards an interactive usage of the final system, a generator component in the Reference Frame concept is not foreseen: the user is expected to create and modify the visual typed graphs.

One question worth discussing is the domain of the mappings contained in a Reference Frame $\mathcal{R}$. As mentioned above, the mappings $(C, Ip, \mathcal{S})$ can depend on type information contained in $(\mathcal{N}(\mathcal{R}), \mathcal{E}(\mathcal{R}))$ - yet, the domains of the mappings are visual typed graphs of arbitrary types - e.g., a constraint could simply disallow any kind of circle in a graph. Obviously, an alternative would be a focus on *exactly* those structures that can be generated with elements from $\mathcal{N}(\mathcal{R})$ and $\mathcal{E}(\mathcal{R})$. However, there are a number of reasons that speak against this alternative:

Generic Interpretation. The chosen approach principally enables interpretation and constraint mappings defined only over abstract graph structures and visual attributes, without taking into account node or edge type information. This allows e.g., for generic graph algorithms (cf. subsection 2.1.2) or specific layout analysis mechanisms to be formulated easily in terms of Basic Reference Frames.

Closeness of Languages. With the presented constraint mechanism, the ability to refer to elements that are *external* to a specific language is the key to easily express the closeness of that language. Therefore, the option of including entities that are beyond the scope of a certain Basic Reference Frame in constraint predicates or interpretations is an important means. It allows, e.g., the expression of the rule "A node in a calculation tree must not be connected to ANY element outside this domain".

Interoperability. The openness concerning the exact scope of C, Ip, and $\mathcal{S}$ may contribute to interoperability between Reference Frames, as it principally allows a Basic Reference Frame to contain mappings that address elements contained in other Basic Reference Frames.

Considering the theory review as given in chapter 2, the Reference Frame approach as presented can be conceived as a mixture between a metamodeling approach and a visual language specification format rather than a model integration technique: for the latter, an explicit formalism for shared semantics would be needed. However, chapter 6 shows that the formal framework can support semantic

interoperability between languages. Section 4.7 contains a further discussion of the approach and its relations to relevant theoretical fields.

Based on the concept of Basic Reference Frames, the following parts of this section describe several approaches how to enable interoperability between Reference Frames. Here, a first helpful criterion is whether two Basic Reference Frames contain the same node or edge types:

Definition 4.11 *A set S of Basic Reference Frames is called* unambiguous *if no node or edge type is contained in more than one of the Basic Reference Frames:*

- $\forall \mathcal{R}_1, \mathcal{R}_2 \in S : (n \in \mathcal{N}(\mathcal{R}_1) \wedge n \in \mathcal{N}(\mathcal{R}_2)) \Rightarrow \mathcal{R}_1 = \mathcal{R}_2$
- $\forall \mathcal{R}_1, \mathcal{R}_2 \in S : (e \in \mathcal{E}(\mathcal{R}_1) \wedge e \in \mathcal{E}(\mathcal{R}_2)) \Rightarrow \mathcal{R}_1 = \mathcal{R}_2$

At first sight, this criterion seems to severely restrict interoperability between Reference Frames, since an unambiguous set of Reference Frames does not allow the sharing of any primitives. Yet, unambiguous Reference Frame sets do also have some advantages which justify their special treatment:

Unique mapping. Unambiguity allows for a deterministic way of relating element types to Basic Reference Frames. A number of functions on the implementation level like, e.g., the calculation of synchronization contexts, rely on this bijective relation.

Avoidance of conflicts. Interoperability between two Reference Frames is hard to achieve if these contain node or edge types with the same identification, but with different meaning associated to them. Even on the theoretical level, the arising ambiguities of such an approach are obvious. On an implementation level, conflicting node or edge type definitions (which would translate to conflicting class definitions) are even harder to solve.

Despite these advantages, the isolation that unambiguous Basic Reference Frame sets induce is in conflict with the intention of interoperability. In the following, I introduce two methods that make up for this. Instead of allowing for element types to be contained in multiple Basic Reference Frames and thereby offering *implicit* interoperability, I make use of *explicit* relations which declare the use of external elements.

4.5.1 Import Relations

One simple approach that allows for interoperability between Reference Frames, and that at the same time retains the bijective relation between Basic Reference Frames and their defined node and edge types is the use of type imports operators. This allows Reference Frames to explicitly refer to externally defined node or edge types and thus re-use them apart from their original contexts.

Definition 4.12 *Let $\mathcal{N}$ and $\mathcal{N}'$ denote sets of node types and $\mathcal{E}$ and $\mathcal{E}'$ sets of edge types with $\mathcal{N} \cap \mathcal{N}' = \emptyset$ and $\mathcal{E} \cap \mathcal{E}' = \emptyset$, and let V_N and V_E be visual node and edge attributes. For a set C of constraint mappings, a semantic domain D, a semantic mapping Ip, and a generic synchronization context mapping S, the structure*

$$\langle \mathcal{N}, \mathcal{E}, V_N, V_E, C, D, Ip, \mathcal{S} \rangle^{\swarrow \langle \mathcal{N}', \mathcal{E}' \rangle}$$

is called an Interoperable Reference Frame that imports $\mathcal{N}'$ and $\mathcal{E}'$.
A multiple application of the import operator is permitted, and is defined by:

$$\left(\mathcal{R}^{\swarrow \langle \mathcal{N}', \mathcal{E}' \rangle}\right)^{\swarrow \langle \mathcal{N}'', \mathcal{E}'' \rangle} := \mathcal{R}^{\swarrow \langle \mathcal{N}' \cup \mathcal{N}'', \mathcal{E}' \cup \mathcal{E}'' \rangle}$$

For Interoperable Reference Frames, the same remarks about the interrelations of elements as given for Basic Reference Frames (cf. definition 4.10) apply: the only exception to this is that the constraints and semantics mappings may make use of the type information provided by the imported types $\mathcal{N}'$ and $\mathcal{E}'$, and their subtypes as contained in definition 4.2.

Any Basic Reference Frame can of course also be conceived as an Interoperable Reference Frame with empty import sets. For this reason, the general term Reference Frame will be used as an integrative notion which covers both Basic and Interoperable Reference Frames.

Definition 4.12 allows the expression of Reference Frames that know more node and edge types than they define themselves, without introducing ambiguities. If a set of Basic Reference Frames is unambiguous, the import operation is always well-defined in the sense that imported and defined elements are disjunct:

Proposition 4.3 *Let $S=\bigcup_{i\in I}\mathcal{R}_i$ be a unambiguous set of Basic Reference Frames. Then with any $\widehat{\mathcal{R}}\in S$, $\mathcal{N}'\subseteq\bigcup_{i\in I}\mathcal{N}(\mathcal{R}_i)\setminus\mathcal{N}(\widehat{\mathcal{R}})$, and $\mathcal{E}'\subseteq\bigcup_{i\in I}\mathcal{E}(\mathcal{R}_i)\setminus\mathcal{E}(\widehat{\mathcal{R}})$, the Interoperable Reference Frame $\mathcal{R}':=\widehat{\mathcal{R}}^{\swarrow\langle\mathcal{N}',\mathcal{E}'\rangle}$ is well-defined.*

Proof. Due to the unambiguity property of S, $\mathcal{N}(\widehat{\mathcal{R}})\cap\mathcal{N}(\mathcal{R}_i)=\emptyset$ for all $\mathcal{R}_i\in S\setminus\widehat{\mathcal{R}}$. Consequently, we have $\bigcup_{i\in I}\mathcal{N}(\mathcal{R}_i)\setminus\mathcal{N}(\widehat{\mathcal{R}})=\emptyset$, and therefore also, by definition of $\mathcal{N}'$, the required relation $\mathcal{N}'\cap\mathcal{N}(\widehat{\mathcal{R}})=\emptyset$. The same holds for the edge types.

Though unambiguous sets of Basic Reference Frames offer the option of arbitrary imports without destroying the unambiguity criterion, one important question is under which conditions these imports do actually make sense, considering that Reference Frames are abstractions of modeling languages, and thus potentially "self-contained" in some sense. In formal notation, this is the following question: if $\mathcal{R}$ is a Basic Reference Frame (and thus the elements of $\mathcal{R}$ only defined in terms of the types $\mathcal{N}(\mathcal{R})$ and $\mathcal{E}(\mathcal{R})$), does then an Interoperable Reference Frame $\mathcal{R}^{\swarrow\langle\mathcal{N}',\mathcal{E}'\rangle}$ make sense? This can be dealt with on the level of the elements that constitute an Interoperable Reference Frame.

- The node and edge attributes V_N and V_E are defined independent of specific Reference Frames anyway, so that an import of node or edge types is not of relevance for the layout mappings.
- The constraint mappings set C, the semantics (D,*Ip*), and the generic synchronization context mapping $\mathcal{S}$ are more problematic. In general, it is not guaranteed that C (which was originally designed for a smaller scope) preserves syntactic correctness of visual typed graphs over extended sets of node types. As a consequence, a coherent semantics cannot be expected. Finally, $\mathcal{S}$ will usually not be able to generate a suitable synchronization context for the imported element types.

These drawbacks are serious, but could be expected: a simple element import of elements from one modeling language into another one can in most cases not be possible without any add-on work. There are, however, some situations in which a simple type import does indeed make sense. If no constraints are necessary to preserve syntactic consistency (neither because of $\mathcal{R}$ nor because of the imported types), no formal semantics of the imported elements is available (e.g. in their origin Basic Reference Frame), and a trivial synchronization context (e.g., single node sharing allowed) for the imported types is reasonable, then the type import is generally unproblematic. Here, an example is the import of "generic" elements into specialized modeling languages. E.g., if a Basic Reference Frame $\mathcal{R}$ describes a formal modeling language and another one, $\mathcal{R}'$, contains a set of "comment" nodes

and edges used to simply put in arbitrary text, then the import of some of the comment elements into $\mathcal{R}$ is usually possible without problems and results in an Interoperable Reference Frame which offers not only the elements for the construction of formal models, but also offers the imported means for comments. Advanced and more intelligent approaches are of course imaginable (e.g., the consideration of the synchronization context mappings defined in the "original" Reference Frame that defines the imported types), but are not possible within the "import" mechanism. The next subsection presents an example of a more advanced approach which allows the modification of synchronization contexts along with the import of node and edge types.

Apart from the examples presented in this subsection, which illustrate how Basic Reference Frames can serve as foundations for reasonable Interoperable Reference Frames, also scenarios with inherently interoperable Reference Frames (i.e., not based on a corresponding Basic Reference Frame) are imaginable. Here, examples include the explicit import of generic visualizer components for specific data types (e.g., tables for arrays of numbers) into an Interoperable Reference Frame with the aim of allowing the display of model states without having to define the displays themselves. In subsection 6.3.2, a detailed example for this (the inclusion of function plotters within a System Dynamics Reference Frame) is illustrated. This section also addresses some other implementation related aspects. E.g., the type import leads to requirements concerning the encapsulation of components: the "importable" elements (node and edge types) should be designed with suitable interfaces that allow for reuse also technically (e.g., concerning access rights).

4.5.2 Is-a Relations

The previous subsection described a first conceptual way of establishing connections between Basic Reference Frames. The import relation basically enables a Basic Reference Frame to *know other node and edge types*. As outlined, already this very lightweight relation is usable in a considerable number of cases. However, the type import does not allow for expressing direct dependencies and interrelations between Reference Frames. The mechanism as defined in the following addresses this problem: syntactically based on the type import, it enables the formulation of a Reference Frame which extends another one and adds functionality to it.

Definition 4.13 *Let* $\mathcal{R} = \langle \mathcal{N}, \mathcal{E}, V_N, V_E, C, D, Ip, \mathcal{S} \rangle^{\swarrow \langle \mathcal{N}^*, \mathcal{E}^* \rangle}$ *and* $\mathcal{R}' = \langle \mathcal{N}', \mathcal{E}', V'_N, V'_E, C', D', Ip', \mathcal{S}' \rangle^{\swarrow \langle \mathcal{N}'^*, \mathcal{E}'^* \rangle}$ *be Reference Frames. Then* $\mathcal{R}'$ *is called an* extension *of* $\mathcal{R}$*, expressed by the notation* $\mathcal{R}' \nwarrow_{\mathcal{R}}$*, if the following relations hold:*

- $\mathcal{N} \cup \mathcal{N}^* \subseteq \mathcal{N}'^*$ *(i.e.,* $\mathcal{R}'$ *imports all the node types defined or imported by* $\mathcal{R}$*)*
- $\mathcal{E} \cup \mathcal{E}^* \subseteq \mathcal{E}'^*$ *(*$\mathcal{R}'$ *imports all the edge types defined or imported by* $\mathcal{R}$*)*
- $V_N \subseteq V'_N$
- $V_E \subseteq V'_E$
- $C \subseteq C'$ *(i.e.,* C' *contains at least all the mappings contained in* C*)*
- *There is a set* D_E *so that* $D' = D \times D_E$
- *domain(Ip)* $\subseteq$ *domain(*Ip'*)*

According to this definition, a Reference Frame $\mathcal{R}' \nwarrow_{\mathcal{R}}$ does not only import the primitives (node and edge types) of $\mathcal{R}$, but also retains the visual attributes, the set of constraint mappings, and the domain of the semantic mapping. $\mathcal{R}'$ is

allowed to extend all these sets. The semantic domain of $\mathcal{R}'$ is also an extension of the one of $\mathcal{R}$. The formal notation of the cross product models better (compared to a superset relation) the additional semantic attributes that $\mathcal{R}'$ can define: these constitute D_E. The chosen approach expresses that the semantic domain defined by $\mathcal{R}'$ fully retains the original set D and thereby allows the original semantic mapping to be included as a part of the extended mapping Ip', but at the same time new attributes are possible independently of the original ones. Of course, a Reference Frame extension without really changing the semantic domain is possible: here, a trivial extension set D_E (consisting, e.g, of one neutral element) can be used.

It does not make sense to include a specific requirement about the synchronization context mapping into the above definition, as these mappings rely on the types defined by the Reference Frames. These, however, are usually disjunct (at least for unambiguous Reference Frame sets, cf. definition 4.11).

Though a Reference Frame that extends another one clearly takes the extended one as a base for its own components, definition 4.13 does not guarantee preservation of syntax or semantics of visual typed graphs. For the case of the constraint mappings, this is expressed through the subset relation between the set defined by the base Reference Frame and the set defined by the extension Reference Frame. An immediate consequence of this is that the set of "correct" expressions can be reduced (but not widened!) through the extension. The semantic mapping may even completely change. This openness of the extension mechanism generally makes sense, as some examples will illustrate later. It does, however, also allow for inconsistent sets of constraint mappings sets, as the next proposition outlines:

Proposition 4.4 *If $\mathcal{R}$, $\mathcal{R}'$ and $\mathcal{R}''$ are Reference Frames with ${\mathcal{R}'}_{\nwarrow\mathcal{R}}$ and ${\mathcal{R}''}_{\nwarrow\mathcal{R}}$, then the conformity of a visual typed graph with respect to $\mathcal{C}(\mathcal{R}')$ does not imply conformity with respect to $\mathcal{C}(\mathcal{R}'')$, and vice versa.*

Proof. Consider the example case of $C(\mathcal{R}') = C(\mathcal{R})$ and $C(\mathcal{R}'') = C(\mathcal{R}) \cup \{c\}$ so that there is at least one visual typed graph G which is conform with $C(\mathcal{R})$ but not with c. Then G is correct in terms of $\mathcal{C}(\mathcal{R}')$ (and in terms of $\mathcal{C}(\mathcal{R})$), but not in terms of $\mathcal{C}(\mathcal{R}'')$.

As shown, the extension mechanism of Reference Frames may lead to a diffusion of the overall term "syntactic correctness": inconsistencies are not prevented. In the simplest case (one Reference Frame extending another one), these inconsistencies might easily be overcome by defining the constraint mappings set of the base Reference Frame as "prior" to its extensions. However, the case presented in proposition 4.4 ($\mathcal{R}'$ and $\mathcal{R}''$ are independent extensions that are in conflict with each other) underlines the basic problem: Based upon multiple Reference Frames, it is difficult for a framework to define a global notion of "syntactic correctness" that fully takes into account all definitions and constraints. As illustrated, this can even be the case for closely related Reference Frames. A simple merge of the constraint mappings sets, resulting in a perhaps too narrow set of "correct" visual typed graphs, seems to be the only feasible solution for these situations.

One aspect worth mentioning is that the extension mechanism for Reference Frames relies on the fact that the syntax constraints of the base are retained in the extension. This does not allow for declaring "specialized" Reference Frames which allow *more* than their origin. Typically, this approach meets well practical requirements: a modeling language defines certain syntactic rules, and a subtype of the language will usually retain these. The relation between UML2 allowing generally more than UML1 (and, in addition, assigning a different semantics to elements that included also in UML1) is a counterexample here - yet, one might argue whether in this case, UML2 can still be called a *specialization* of UML1, or whether UML1 and UML2 are simply different languages.

In contrast to the above observations that are related to extended Reference Frames modifying the integrity constraints sets of their base concerning the basic element types, a number of realistic extensions of modeling languages are likely to preserve syntax and semantics of their foundations. These extensions are characterized in the following definitions:

Definition 4.14 *If $\mathcal{R}$ and $\mathcal{R}'$ are Reference Frames so that $\mathcal{R}' \nwarrow_{\mathcal{R}}$, then the extension is called* syntactically consistent *if $\mathcal{R}'$ preserves the syntax of $\mathcal{R}$, i.e. if all visual typed graphs which consist only of node and edge types defined in $\mathcal{R}$ and that are syntactically correct in terms of $\mathcal{C}(\mathcal{R})$ are also correct in terms of $\mathcal{C}(\mathcal{R}')$.*

Definition 4.15 *If $\mathcal{R}$ and $\mathcal{R}'$ are Reference Frames so that $\mathcal{R}' \nwarrow_{\mathcal{R}}$, then the extension is called* semantically consistent *if it is syntactically consistent, and if $\mathcal{R}'$ in addition preserves the semantics of $\mathcal{R}$, i.e. if for a visual typed graph G that is syntactically correct according to $\mathcal{C}(\mathcal{R}')$, the semantics should be retained:*

$$Ip(\mathcal{R}')(G) = (Ip(\mathcal{R})(G), d) \text{ for some } d \in D_E \text{ , if } D(\mathcal{R}') = D(\mathcal{R}) \times D_E$$

The following proposition shows that under certain conditions, the syntactic integrity constraints of the base Reference Frame are the same as the ones of the extended Reference Frame.

Proposition 4.5 *If $\mathcal{R}$ and $\mathcal{R}'$ are Reference Frames so that the extension $\mathcal{R}' \nwarrow_{\mathcal{R}}$ is syntactically consistent, then for visual typed graphs which consist only of node and edge types defined in $\mathcal{R}$, syntactic correctness in terms of $\mathcal{C}(\mathcal{R})$ is equivalent to syntactic correctness in terms of $\mathcal{C}(\mathcal{R}')$.*

Proof. An immediate consequence of $\mathcal{R}'$ extending $\mathcal{R}$ is that $C(\mathcal{R}) \subseteq C(\mathcal{R}')$. Therefore, the syntactic correctness in terms of $\mathcal{C}(\mathcal{R}')$ generally implies correctness in terms of $\mathcal{C}(\mathcal{R})$. For the specific case of visual typed graphs which consist only of node and edge types defined in $\mathcal{R}$, the specific characteristics of syntactically consistent extensions (according to definition 4.14) show the other direction.

An obvious result of proposition 4.5 is that inconsistent constraint mapping sets as constructed in the proof of proposition 4.4 can not occur for syntactically consistent Reference Frame extensions. However, the general problem remains, though restricted to a special class of visual typed graphs.

Definition 4.16 *A visual typed graph $\langle G, \mathcal{N}, \mathcal{E}, L \rangle$ is called* heterogeneous *with respect to a set $\{\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_n\}$ of Reference Frames if there is no $i \in \{1, \ldots, n\}$ with $\mathcal{N} \subseteq \mathcal{N}(\mathcal{R}_i)$ and $\mathcal{E} \subseteq \mathcal{E}(\mathcal{R}_i)$.*

Proposition 4.6 *Proposition 4.4 also holds if the extensions are syntactically consistent.*

Proof. Let n be a node type not contained in $\mathcal{N}(\mathcal{R})$ but imported into both $\mathcal{R}'$ and $\mathcal{R}''$. Let G be a visual typed graph which contains nodes of type n. Consider the example case of $C(\mathcal{R}') = C(\mathcal{R}) = \emptyset$, and $C(\mathcal{R}'') = \{c\}$ with G not conform with c. Then G is correct in terms of $\mathcal{C}(\mathcal{R}')$ (and in terms of $\mathcal{C}(\mathcal{R})$), but not in terms of $\mathcal{C}(\mathcal{R}'')$.

The proof of the previous proposition needed external elements to construct an inconsistent set of constraint mappings. With elements coming from the original set of primitive entities as defined in the base Reference Frame, such a construction

is not possible. Furthermore, the proof needed the property that a heterogeneous visual typed graph is correct in terms of the basic Reference Frame $\mathcal{R}$ (as the extensions $\mathcal{R}'$ and $\mathcal{R}''$ can only narrow notion of syntactic correctness, not widen it). This correctness of the heterogeneous structure, however, can be prevented with appropriate integrity constraint mappings in $\mathcal{C}(\mathcal{R})$ (cf. section 4.2). This offers a means to describe "encapsulated" languages which protect their notion of syntactic correctness, provided only consistent extensions are used.

The extension mechanism for Reference Frames as presented in this subsection is far more expressive than the inclusion principle shown in the previous subsection. There are a number of Reference Frame extensions that are reasonable for various causes. Obviously, the basic motivation for is-a relations between modeling languages is the reuse of functionality. On the application level, two more specific use cases for the extension mechanism are the following:

The use of generic libraries. Reference Frame extensions allow the access to generic node and edge type libraries - in contrast to simple imports of these types, the whole context (syntactic constraints, semantics, etc.) of these node and edge types can be considered.

Iterative adding of functionality. For several reasons, the function of defining a Reference Frame in several iterative steps might be desired: the option of offering "core" modeling languages that are very intuitive but only have a a restricted functionality, and "advanced" fully expressive versions of these languages may be attractive from both usability and also educational reasons.

Examples for the first category that go beyond node and edge type imports include the re-use of generic graph algorithms that are implemented as semantic mappings, or the definition of certain logging and tracking functions, e.g. for interaction analysis or user feedback, within generic Reference Frames that can then even be defined with empty node and edge type sets, the functionality being a side effect of the interpretation mapping function. These generic Reference Frames can then serve as a base for other Reference Frames that are compatible with this kind of logging or tracking.

A concrete example for the second category (iterative adding of functionality) from the domain of stochastics is shown in figure 6.4 in chapter 6.

4.6 Reference Frame Based Interpretation

The previous section described the concept of a Reference Frame as a formal abstraction of a modeling language that makes use of a graph based representation. A Reference Frame can contain specific syntactic integrity constraints, and has a semantic mapping associated. Thus, it is able to *interpret* visual typed graphs that consist of (at least) "his" node and edge types. However, the question of interpretation has not been fully addressed up to now:

- What does interpretation of a visual typed graphs by *multiple* Reference Frames mean?
- How can syntax and semantics of *heterogeneous* visual typed graphs be defined?

Obviously, both questions are important to address with respect to the desired interoperability. However, this section of the thesis does not show *how* an interpretation of heterogeneous visual typed graphs by multiple Reference Frames can practically be done (this will be done in the next chapters). Instead, I want to

prepare the implementation parts with some theoretical remarks that outline *what* interpretation under the mentioned constraints may be. A first version of these conceptual approaches has already been published (Pinkwart, 2003).

Concerning syntax, there is not much choice. According to the definitions 4.6 and 4.10, a visual typed graph G is syntactically correct with respect to a Reference Frame $\mathcal{R}$ if all the constraint mappings in $C(\mathcal{R})$ deliver `true` for G. The domain of the semantic mapping of $\mathcal{R}$ consists only of those visual typed graphs that are syntactically correct concerning $C(\mathcal{R})$. To allow for an interpretation of a visual typed graph G by multiple Reference Frames $\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_n$ at all, G will necessarily have to be conform with *all* the constraint mappings contained in $\bigcup_{i=1}^{n} C(\mathcal{R}_i)$. A trivial consequence of this is that syntactic correctness concerning one Reference Frame $\mathcal{R}$ does not guarantee correctness concerning a set of Reference Frames, even if the set contains $\mathcal{R}$ and G consists only of node and edge types contained in $\mathcal{N}(\mathcal{R})$ and $\mathcal{E}(\mathcal{R})$.

While the syntactic aspects can be treated generically as shown above (with the identified drawbacks), the case is much more unclear and harder for the semantics of visual typed graph. For a syntactically correct (concerning Reference Frames $\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_n$) visual typed graph G, each Reference Frame has its own interpretation of G, namely the image of the semantic mapping $Ip(\mathcal{R})(G)$. A straightforward approach for the definition of an interpretation based on $\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_n$ would therefore be the use of the cross product, with the integrated semantic mapping having the image $D(\mathcal{R}_1) \times D(\mathcal{R}_2) \times \ldots \times D(\mathcal{R}_n)$. The semantics of a visual typed graph would then be a tuple that consists of the individual interpretations done by the participating Reference Frames. This, however, has three disadvantages.

- Such a solution does not take into account interoperability between Reference Frames at all: the abstraction does not foresee interrelations between semantics.
- The pure cross product notation does not represent the finest possible granularity, as it does not take into account the specific relations between Reference Frames and heterogeneous visual typed graphs.
- The straightforward approach causes the semantics of a visual typed graph to change if a Reference Frame is added to the set of "interpreters", even if the added Reference Frame does not have anything to do with the graph.

These reasons motivate another approach, the implementation of which will be shown in chapter 6. The approach makes use of an interpretation scheme which logically splits a heterogeneous visual typed graph into several subgraphs, and foresees interpretations of these subgraphs by sets of Reference Frames.

4.6.1 Substructures of Visual Typed Graphs

The conceptual approaches presented in the following three subsections use visual typed graphs (cf. definition 4.5) as the primary subject of interpretation. The case of multiple graphs that are in some kind of relation to each other is not treated. The only exception to this is that synchronized graph instances are generically covered by means of the synchronization contexts - the inclusion of these is presented in the following chapters of this thesis.

Apart from visual typed graphs as basic data structures to be interpreted, another assumption made in the following is that this visual typed graph $\langle G, \mathcal{N}, \mathcal{E}, L \rangle$ is interpreted by a set R of Reference Frames which fulfils the unambiguity criterion expressed in definition 4.11. With the import and is-a relations as presented in section 4.5 (and other imaginable ones, based on the concept of Reference Frames), this does not limit the scope of the following approach too much.

Definition 4.17 *Let $\mathcal{R}^{\swarrow\langle\mathcal{N}',\mathcal{E}'\rangle}$ be a Reference Frame. Then the union $\mathcal{N}(\mathcal{R}) \cup \mathcal{E}(\mathcal{R})$ is called* DEFINES*($\mathcal{R}$).*

If a Reference Frame $\mathcal{R}$ is contained in an unambiguous set of Reference Frames R, DEFINES($\mathcal{R}$) characterizes the element types that $\mathcal{R}$ contributes to the types contained in R, and there is a mapping between types and defining structures: If x is an arbitrary node or edge type contained in $\mathcal{N}$ or $\mathcal{E}$, then the unambiguity property of R ensures that there is at most one Reference Frame $\mathcal{R} \in R$ which defines x, i.e. where $x \in$ DEFINES($\mathcal{R}$).

The DEFINES set can thus be used for expressing to which Reference Frame (out of a set of available ones) a certain node or edge type "belongs". This relation will be important in the implementation parts. However, it does not cover the interoperability mechanisms between Reference Frames, in particular the import and is-a relations. To incorporate this, the introduction of a second concept is necessary:

Definition 4.18 *If $\mathcal{R}^{\swarrow\langle\mathcal{N}',\mathcal{E}'\rangle}$ is a Reference Frame, then the union* DEFINES*($\mathcal{R}$) $\cup\mathcal{N}' \cup \mathcal{E}'$ is called* KNOWS*($\mathcal{R}$).*

As defined, the set KNOWS($\mathcal{R}$) includes all the node and edge types that a Reference Frame imports and thus, loosely speaking, the types that it has detailed information about and can semantically access.

By definition, we have the relation DEFINES($\mathcal{R}$) $\subseteq$ KNOWS($\mathcal{R}$), with equality only if $\mathcal{R}$ does not import any types. Another immediate conclusion from definitions 4.13 and 4.18 is that for a Reference Frame extension $\mathcal{R}'_{\nwarrow\mathcal{R}}$, the relation KNOWS($\mathcal{R}$) $\subseteq$ KNOWS($\mathcal{R}'$) holds.

Now, let us consider a visual typed graph $\langle G, \mathcal{N}, \mathcal{E}, L\rangle$ and a Reference Frame $\mathcal{R}$. Then the largest substructure of G to which $\mathcal{R}$ has detailed access in the sense that all the contained (node or edge) types are imported or defined by $\mathcal{R}$ is obviously determined by the set KNOWS($\mathcal{R}$):

Definition 4.19 *Let $\langle G, \mathcal{N}, \mathcal{E}, L\rangle$ be a visual typed graph, and $\mathcal{R}$ a Reference Frame. Then $G_{|\mathcal{R}}$ denotes the largest subgraph of G whose node and edge domains are subsets of* KNOWS*($\mathcal{R}$). This subgraph is called* induced by $\mathcal{R}$.

Here, the *subgraph* relation between G and $G_{|\mathcal{R}}$ instead of a *subset* relation is used, as otherwise the problem of "dangling" edges could occur: if an edge type is in the KNOWS set, but not the type of an adjacent node, an inclusion of the edge into $G_{|\mathcal{R}}$ would destroy the graph structure. The proposed approach avoids this problem, but therefore potentially includes slightly too small substructures of G into $G_{|\mathcal{R}}$.

An immediate result of definition 4.19 is that there is no direct relation between a node or edge in a graph, and the Reference Frames knowing this node/edge, as the following proposition illustrates:

Proposition 4.7 *Let $\langle G, \mathcal{N}, \mathcal{E}, L\rangle$ be a visual typed graph, and let S be a set of Reference Frames. If φ denotes the mapping between a node n of G and the set of Reference Frames whose induced subgraph of G contains n (thus φ being a mapping from N(G) to $\mathcal{P}(R)$), then the following relationships hold:*

1. *φ may map a node to the empty set*
2. *In the general case, φ maps all nodes to sets with one or zero elements only if R is unambiguous and no Reference Frame in S imports an element that is defined or imported by another Reference Frame in S.*

3. *In general, φ is not injective.*

4. *φ is usually not surjective. It is not even guaranteed that each Reference Frame of S is contained in one of the sets of image(φ))*

Proof.

1. If n is a node of a type that is not contained in the KNOWS set of any Reference Frame contained in S, then n will not appear in any of the induced subgraphs. Consequently, $\varphi(n) = \emptyset$.

2. If S is unambiguous, the DEFINES sets are disjoint. Under the condition as expressed above, also the imported sets are disjoint from each other and from the DEFINES sets, so that finally also the KNOWS sets are disjoint. Consequently, no node can be contained in two subgraphs that are induced by Reference Frames in S.

3. In fact, φ is almost never injective: if, e.g., G contains two nodes n_1 and n_2 of the same type, then obviously $\varphi(n_1) = \varphi(n_2)$.

4. If S contains a Reference Frame $\mathcal{R}$ with KNOWS($\mathcal{R}$) not containing any of the types contained in G, then $\mathcal{R}$ will not be contained in any of the images that φ produces. In particular, a concrete counterexample is an empty graph G and a non-empty set S. Here, $\{S\} \notin image(\varphi)$.

The previous proposition outlined that, given a visual typed graph, it is not trivial to determine the "semantically near" subgraph with respect to the interpretation mechanism and scope of a Reference Frame. These problems are caused by the possibility of importing elements and make an implementation challenging: a central "management" component could of course manage the relations between nodes and edges contained in a visual typed graph on the one hand and Reference Frames on the other hand (by simply considering the KNOWS and DEFINES sets), but further interpretation functions can not be delegated to such a central component.

Together with chapter 6, the next two subsections show how a solution can be based on different interpretation types and their aggregation.

4.6.2 Interpretation Types

As defined in section 4.5, each Reference Frame $\mathcal{R}$ can principally interpret any visual typed graph, irrespective of whether the Reference Frame really relates to the graph and its types. Formally speaking, $Ip(\mathcal{R})$ is not limited to structures of types contained in KNOWS($\mathcal{R}$). However, as motivated in the annotations to definition 4.10, the mapping Ip can (usually) not depend on types not accessible to $\mathcal{R}$. To solve this conflict of keeping the interpretation mapping general enough to ensure semantic interoperability, and at the same time making it expressive enough for structures of known types, I define two subtypes of interpretation:

Generic Interpretation. An interpretation of a visual typed graph $\langle G, \mathcal{N}, \mathcal{E}, L\rangle$ by a Reference Frame $\mathcal{R}$ is called *generic*, if the interpretation mapping relies only on the abstract graph structure of G and its layout L. In this case, the interpretation is denoted with $Ip_{gen}(\mathcal{R})(\langle G, \mathcal{N}, \mathcal{E}, L\rangle)$

Domain Specific Interpretation. If an interpretation of a visual typed graph $\langle G, \mathcal{N}, \mathcal{E}, L\rangle$ by a Reference Frame $\mathcal{R}$ makes use of the types $\mathcal{N}$ and $\mathcal{E}$, then this interpretation is called *domain specific*. Such an interpretation is expressed as $Ip_{dom}(\mathcal{R})(\langle G, \mathcal{N}, \mathcal{E}, L\rangle)$.

Proposition 4.8 *Given a heterogeneous visual typed graph G, then $G_{|\mathcal{R}}$ is the largest subgraph of G for which a domain specific interpretation by a Reference Frame $\mathcal{R}$ is possible.*

Proof. $G_{|\mathcal{R}}$ consists only of nodes and edges whose type is contained in KNOWS($\mathcal{R}$). As an immediate consequence, $\mathcal{R}$ can make use of the types for interpretation, and can therefore conduct a domain specific interpretation of $G_{|\mathcal{R}}$. The fact that $G_{|\mathcal{R}}$ is the largest subgraph of G with this characteristic is due to the maximality condition in definition 4.19.

The two types of interpretation (generic and domain specific) will serve as a means of distinction in the following, and they will have natural correspondences in the implementation parts. Proposition 4.8 outlined that the maximum scope for a domain specific interpretation can easily be determined: it is the subgraph that a Reference Frame induces on a visual typed graph. Examples for domain specific interpretation are easy to find (cf. example 4.4). They include most of the algorithms used within modeling languages. The following example contains three different generic interpretations of visual typed graphs and demonstrates that already a generic interpretation can indeed be very useful in the target domain of collaborative modeling.

Example 4.6 *Examples for generic interpretation mappings include the following:*

- *All the mappings that operate on abstract graph structures can be seen as generic interpretations. This includes trivial ones (like, e.g., the counting of nodes and edges that a graph contains) as well as advanced complex graph algorithms. One imaginable generic interpretation where the semantic domain is the set of visual typed graphs is, e.g., the calculation of a spanning tree.*

- *A generic interpretation may also access layout information. This allows the implementation of puzzle-style layout checks as generic interpretations, as well as routines which search for nodes being contained in others in the spatial arrangement.*

- *A layout may contain visual node and edge attributes that are not only related to the spatial arrangement of elements, but also to the person that created them (e.g., for awareness mechanisms). A generic interpretation that operates on these attributes can, e.g., calculate usage statistics of a visual typed graph, and can thus build the base for feedback mechanisms that are based on these calculations.*

4.6.3 The Integration of Multiple Interpretations

The two previous subsections introduced the notions of induced subgraphs that result from filtering visual typed graphs by Reference Frames, and two kinds of interpretation (generic and domain specific) that a Reference Frame can conduct on a visual typed graph. I have also motivated that subgraphs as induced by Reference Frames are exactly the structures that allow for domain-specific interpretation by the corresponding Reference Frame.

This subsection now adds one more ingredient: the interpretation of a visual typed graph by *multiple* Reference Frames. As this will involve both generic and domain specific interpretations, a small preparation is necessary:

If $\langle G, \mathcal{N}, \mathcal{E}, L \rangle$ is a visual typed graph that is interpreted by a Reference Frame $\mathcal{R}$ with interpretation mapping $Ip(\mathcal{R})$, then (using the terms as introduced in subsection 4.6.2), the interpretation is:

$$Ip(\mathcal{R})(\langle G,\mathcal{N},\mathcal{E},L\rangle) = Ip_{gen}(\mathcal{R})(\langle G,\mathcal{N},\mathcal{E},L\rangle) \otimes Ip_{dom}(\mathcal{R})(\langle G_{|\mathcal{R}},\mathcal{N},\mathcal{E},L\rangle)$$

As stated in the previous subsection, the generic interpretation does not make use of the type information associated to nodes and edges. Therefore, $\mathcal{N}$ and $\mathcal{E}$ can be replaced with anonymous variables (or even the empty set) at the respective location in the formula to emphasize the independency of Ip_{gen} from $\mathcal{N}$ and $\mathcal{E}$, and the result is:

$$Ip(\mathcal{R})(\langle G,\mathcal{N},\mathcal{E},L\rangle) = Ip_{gen}(\mathcal{R})(\langle G,\_,\_,L\rangle) \otimes Ip_{dom}(\mathcal{R})(\langle G_{|\mathcal{R}},\mathcal{N},\mathcal{E},L\rangle)$$

Now, this result can be extended towards multiple Reference Frames. If a set $R = \{\mathcal{R}_1,\mathcal{R}_2,\ldots\}$ of Reference Frames and a visual typed graph $\langle G,\mathcal{N},\mathcal{E},L\rangle$ are given, then an interpretation of the visual typed graph by the set can be defined as the aggregation of the different interpretations as conducted by the single Reference Frames:

$$Ip(\langle G,\mathcal{N},\mathcal{E},L\rangle) := \bigotimes_{\mathcal{R}\in R} Ip(\mathcal{R})(\langle G,\mathcal{N},\mathcal{E},L\rangle)$$

As shown above, the left part of the relation can be expressed in further detail and results in:

$$Ip(\langle G,\mathcal{N},\mathcal{E},L\rangle) = \bigotimes_{\mathcal{R}\in R} \left(Ip_{gen}(\mathcal{R})(\langle G,\_,\_,L\rangle) \otimes Ip_{dom}(\mathcal{R})(\langle G_{|\mathcal{R}},\mathcal{N},\mathcal{E},L\rangle)\right)$$

An aspect not mentioned up to now is the exact meaning of the aggregation operator $\otimes$ in the formulas above. This was done on purpose, as the shown relations are independent of the concrete way this operator works. Basically, the following two different principles for the implementation of $\otimes$ can be distinguished:

Separation. A first approach is to keep different interpretations of a visual typed graph separate from each other. Here, the aggregation result can be represented as a tuple:

$$Ip_1(\langle G,\mathcal{N},\mathcal{E},L\rangle) \otimes Ip_2(\langle G,\mathcal{N},\mathcal{E},L\rangle) := \langle Ip_1(\langle G,\mathcal{N},\mathcal{E},L\rangle), Ip_2(\langle G,\mathcal{N},\mathcal{E},L\rangle)\rangle$$

Integration. Another approach is to allow the different aggregated interpretations to be really integrated and combined in some way, including the acceptance of side effects. In this case, a suitable notation format makes use of a merging function μ:

$$Ip_1(\langle G,\mathcal{N},\mathcal{E},L\rangle) \otimes Ip_2(\langle G,\mathcal{N},\mathcal{E},L\rangle) := \mu(Ip_1(\langle G,\mathcal{N},\mathcal{E},L\rangle), Ip_2(\langle G,\mathcal{N},\mathcal{E},L\rangle))$$

Although the separative approach is formally a specific case of the more general integrative approach (with μ mapping the unchanged interpretation results to a tuple), it makes sense to distinguish the two outlined cases: for practical implementations, there are major differences between the two cases.

The first assigns each Reference Frame "his" component in the interpretation result tuple. Through this one-to-one mapping, a Reference Frame is the unique source for a part of the general interpretation outcome. Furthermore, the first definition of the aggregation is closed in the sense that it does not make use of any further functions (like the μ in the second one), and this way allows for a (relatively) straightforward implementation. However, the strict separation of interpretation results is not a contribution to semantic interoperability.

The second approach better addresses this requirement. With the additional merging function, it allows access to interpretation results across Reference Frames. The following two examples illustrate this:

- It is possible to base the interpretation of a visual typed graph by Reference Frame $\mathcal{R}_2$ on the interpretation that another Reference Frame $\mathcal{R}_1$ did. This enables advanced re-use of results: $\mathcal{R}_1$ can, e.g., be associated to a modeling language, with the interpretation done by $\mathcal{R}_1$ representing calculations required by this language. Reference Frame $\mathcal{R}_2$ could then, e.g., be a model analyzer that checks the model for some kind of correctness (according to some task, or other criteria). This obviously requires (or induces) an order among the interpretations - formally, $Ip_{\mathcal{R}_2}$ here operates on $Ip_{\mathcal{R}_1}(G)$, not on G directly. Disadvantages and inherent problem of this are discussed in the following. However, an advantage is that *external* interpretations (which do not necessarily have to accept visual typed graphs as input) can be integrated. In addition, as long as an implementation of the Reference Frame approach ensures that the results of the semantic mapping are stored associated to the graph that is interpreted, the parameter type problem can not occur in practice.

- The case listed above involves one Reference Frame that takes up the interpretation that another one conducted as input. Another form of interoperability that goes even beyond this unidirectional re-use is illustrated in figure 4.4. Here, a Petri Net (on the left side) interacts with a System Dynamics network (on the right side). The general interpretation includes the two single model interpretations in two respects: the number of tokens in the "amount" place serves as an input for the rate in the System Dynamics net, and vice versa the capacity of the amount place is controlled by the result stock in the System Dynamics net. This shown type of integration obviously requires a non-trivial merging function μ.

Another advantage of the integrative way of interpretation is its high degree of flexibility and expressiveness: the calculi that are used for the interpretation functions Ip and the merging function μ are the only means that limit the expressiveness of the integrated interpretation. If μ is implemented in the same language as the original single interpretations, a loss of expressiveness will therefore not occur.

There are, however, also a number of drawbacks of an integrated interpretation approach as opposed to the separated variant.

- The integrated version obviously adds a further degree of complexity.

- Unless μ is commutative, the order of the aggregated interpretations of a visual typed graph has an impact on the overall interpretation result. This might be critical, as there is no simple way of deciding the order in which to apply the interpretation mappings multiple Reference Frames on a visual typed graph. Even node-wise, this is not evident: of course, the (unique) Reference Frame which defines the node type could get a priority, but how could a (reasonable!) order among the others be determined, given that even multiple imports and extensions are possible?

- Under the premise of semantic interoperability, conflicts between interpretations are possible and inevitable. As any Reference Frame may interpret any visual typed graph, problematic situations like the following two might occur:

  - A Reference Frame $\mathcal{R}_1$ is designed to ensure, e.g., a tree structure of graphs, and, in the case of detecting a cycle, mirrors back these cycles as errors to the user (via an implementation of the Reference Frame that displays the result of the interpretation on the screen). Another Reference Frame $\mathcal{R}_2$ offers the nodes and edge types used for Petri Nets.

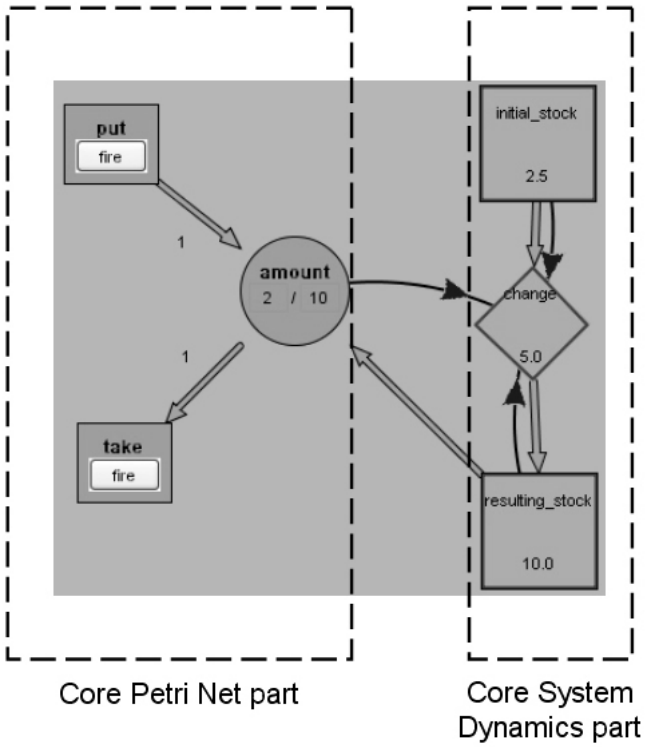

Figure 4.4: Illustration of integrative model interpretation: Petri Nets and System Dynamics

An interpretation of a visual typed graph by these Reference Frames would result in undesired cycle restrictions for Petri Nets.

- It is possible to define Reference Frames which take elements out of their context and define a semantics that is inconsistent with the "original" one as specified by the Reference Frame that defined these elements. The integration in figure 4.4 shows a potential situation where this problem may occur: if the integration between the modeling languages is not defined consistent with the original semantics, the merge function might override the activation rule for transitions in Petri Nets and, e.g., declare the "take" transition active if the initial stock contains at least 2 elements. This hypothetical definition would then indeed violate the semantics of the Petri Net part of the graph.

- It depends on μ (and, of course, on the single interpretation mappings), if the overall integrated interpretation exists in the sense that it is computable.

These negative points of the integrated approach have to be taken into consideration seriously. On the other hand, the separative approach does not allow for semantic interoperability at all. In addition, the localization of problems going along with inconsistent semantics is facilitated through the following observation: splitting the domain specific interpretation Ip_{dom} of a Reference Frame into a *core semantics* and an *interoperability semantics* is easily possible in the approach - the core semantics is related to the part of $G_{|\mathcal{R}}$ that consists of types contained in $\textsc{Defines}(\mathcal{R})$ only, and the interoperability semantics is related to the rest of $G_{|\mathcal{R}}$. This separates "real" modeling language semantics from "added" interoperability. Using this approach, only the interoperability semantics is a candidate for potential problems concerning semantic consistency.

Generally speaking, it is not surprising that no simple and generic easy way of integrating multiple interpretations of heterogeneous models exists. An integration of modeling languages (or, even being more restrictive, their semantics), is often difficult already for the specific case of two concretely given languages. A generic integration solution for all graph based visual languages therefore is unrealistic, as existing research results confirm (cf. section 2.3): either the approaches are only conceptual and not suitable for implementation, or the expressiveness of the technique is quite low. The approach as presented in this chapter aims at an intermediary level of detail which does not reduce the number of covered languages, and which is designed to serve as a formal foundation for implementation purposes - the latter not only reduced to the interpretation of visual typed graphs, but in particular also to the computational representation of modeling languages (Reference Frames) as such.

4.7 Interoperability Issues and Design Aims Met

This chapter proposed a conceptual formal framework whose foundations are the notions of visual typed graphs and heterogeneous models. Here, an important input is the field of graph theory: in the abstract base structure, a heterogeneous model is a mathematical graph. This design decision allows arbitrary graph algorithms to work also on the specific structure of heterogeneous models. Compared to existing modeling techniques like MOF (Meta-Object Facility Specification, n.d.) or DSM (Metacase, n.d.), my approach explicitly focuses on graphs and typed visual structures, and therefore is more specific and operational. However, there are some common points with MODL (cf. table 2.2), in particular the inclusion of data types, associations, and syntax constraints, and - on a higher level - the design towards an object oriented implementation.

The concept of a Reference Frame (which is the formal equivalent of a "modeling language") encapsulates all the components that are needed to manipulate and handle visual typed graphs in collaborative contexts: in particular, a Reference Frame can define types of nodes and edges, syntactic and semantic properties, synchronization features, and relations to other Reference Frames (via different mechanisms, two of which have been explored in more detail). In some aspects, my formalization is similar to the work of Wang and Zeevat (1998): their graphical sorts, operations, and predicates have equivalents in the Reference Frame definition. Yet, their work is designed for graphical (in the sense of visual and geometrical) objects only, whereas my approach focuses on graph based structures - the interpretation of these may be completely independent of visual parameters.

Similar to a number of approaches in the field of visual language theory (Marriott & Meyer, 1998; Haarslev, 1999), the Reference Frame approach makes use of constraint predicates in order to flexibly express syntactical requirements. The concepts of syntax and semantics of Reference Frames, the latter relying on the definition of Harel and Rumpe (2004), allow for an implementation using graph grammars , as done by, e.g., Rekers and Schürr (1997) or Kaul (1982), or other transformation oriented techniques like proposed by, e.g., Lara and Vangheluwe (2004), or Cordella et al. (1998). However, the approach is open in the sense that it does not exclusively require or integrate a specific mechanism (at least not on the conceptual level). This design choice was made with respect to the intended flexibility of the system: a prescription of a particular interpretation method for all supported modeling languages is very likely to reduce the set of languages that are supported. Concerning the relations between Reference Frames, I have avoided the development of a general "add" or "merge" mechanism. According to Geoffrion (1989b), these merges will seldom be reasonable, so that a general mechanism does not make sense. Instead, the proposed approach offers some means of interoperability between Reference Frames, allows for a fine degree of control, does not suggest that an arbitrary merge of modeling languages is unproblematic, and still retains a lot of expressiveness (since an "add" can easily be emulated by imports and extensions).

Based on the concepts of visual typed graphs and Reference Frames, this chapter introduced an abstract interpretation method that integrates multiple interpretations of a visual typed graph by a set of Reference Frames, explicitly allowing every Reference Frame to interpret every visual typed graph. With two different types of interpretation (generic and domain specific), the approach takes into account the degree of semantic access that a Reference Frame has to certain structures. Important inputs for my approach are, again, graph theory (in particular the notion of induced subgraphs), and the concept of formal model semantics (Harel & Rumpe, 2004). In the terminology of Dolk and Kottemann (1993), my interoperability approach allows for both definitional integration and procedural integration, as both the import of types across Reference Frames is supported, and also the interpretations can be mixed. Compared to the work of Geoffrion (1989b) in the field of integrated interpretations using structured modeling, the Reference Frame approach has the advantage of not being restricted to cycle-free structures. One point I share with the argumentation of Geoffrion is that a completely automatic integration of modeling languages (or their interpretation mechanisms) is out of reach. Also in my approach, a non-trivial integration requires either the specification of a merging function, or the adaptation of a Reference Frame in order to incorporate external elements.

Two levels of interoperability have been explicitly considered in various parts within this chapter: *syntactic* and *semantic* interoperability.

The core function of syntactic interoperability is provided with the very basic definition of typed graphs: in these structures, it is possible to connect arbitrary

types of nodes with any types of edges. This builds the foundation which allows for heterogeneous models. The design choice for layouts of typed graphs is another contribution to syntactic interoperability. Layouts are independent of types, and therefore allow not only to build heterogeneous models logically, but also to display them in an integrated manner. This is facilitated through shared layout information. Finally, another aspect of syntactic interoperability results from the scope of constraint mappings: the latter can access structures across domain bounds. With this, a systematic way of describing syntactic relations between modeling languages is enabled.

Also semantic interoperability is supported by several design choices in the formal framework.

- The notion of semantics is kept very open and flexible, and it is externalized. This allows the separation of interpretation from concrete representations, and therefore a possible re-use of semantics across Reference Frames.
- The data structure of visual typed graphs (and heterogeneous models, which are defined as visual typed graphs that contain types defined by different Reference Frames) enables not only syntactic interoperability, but also serves as a generic underlying structure which is suited for multiple interpretations. In a way, these heterogenous models thereby represent the mixed semantics.
- The interpretation function of a Reference Frame is principally not restricted to the set of nodes and edges that this Reference Frame "knows": in general, each Reference Frame is able to interpret all visual typed graphs. This does not seem to make much sense understanding a Reference Frame as an encapsulation of a modeling technique only - however, the concept is expressive enough also for further usages that rely on this enhanced scope for the interpretation mapping. Subsection 8.1.4 illustrates some examples.
- Through the import and is-a mechanisms, a Reference Frame interoperability is even explicitly foreseen. These approaches, which will have natural translations on the implementation level (cf. subsection 6.3.2), allow Reference Frames to have overlapping KNOWS sets. In my approach, the latter is the key that allows for really mixed (and not just "coexistent") interpretations.
- The conceptual framework allows for an integrated formulation and formal notation of heterogeneous model structures and their integrated interpretation by multiple Reference Frames.

There are a number of existing solutions for semantic interoperability that can be compared to my approach. The work of Wang and Liu (2003) treats models as black boxes and merely considers their outer interfaces. In contrast to this, my approach offers a higher degree of insight into the models, and also more flexible ways to interconnect models. This holds also when comparing the Reference Frame approach to the semantic interoperability issues presented by McBrien and Poulovassilis (1999). The only means of connection between modeling languages that they foresee are inter-model edges. Though they are able to provide an in-depth specification of these edge types, they do not offer any further methods of integration (and, in addition, do not address formal model interpretation at all). Finally, a distinctive aspect between my work and the structured modeling technique (Geoffrion, 1989a) with respect to semantic interoperability is that structured modeling puts greater restrictions on the types of supported models, and that no mixed or merged interpretation is foreseen in structured modeling. A common point between my approach and the work of Geoffrion, however, is that similar to SML, all the formal and conceptual approaches in this chapter have been developed with

a view towards implementation. The next chapters of this thesis describe a system architecture which puts into practice the Reference Frame approach.

Chapter 5

Existing Technology

The previous chapter of this thesis presented the conceptual Reference Frame approach to support collaborative modeling with heterogeneous graph based representations. It makes sense to build the software implementation of this approach upon two components: a library for graphs, and a communication technology (to technically support collaboration).

Therefore, this chapter reviews the currently existing state-of-art in these fields. For either of the two components, this is done in two steps: first, a preparation is done by identifying criteria (based on the theoretical results from the previous chapter, and specific requirements related to goals of this thesis) that can be used to distinguish between different technologies. In a second step, the currently available technology is shortly presented and evaluated using the criteria list.

5.1 Criteria for Graph Representations

The first reasonable "ingredient" for a modeling framework which relies on graph based representations is of course a library for graphs and their visualizations. The following subsections contain the criteria relevant for the choice of technology. The selection of criteria is guided by three sources of information: the theoretical results from chapter 2, the specifications and design goals from the introduction, and also general aspects of computer science and software engineering.

5.1.1 Supported Structures

In section 2.1, the most important mathematical concepts in the field of graph theory have been mentioned. These were used within the concept of typed graphs (cf. section 4.1). Of course, the scope of a graph framework in this respect is an important criterion. In particular, the following questions are relevant:

- What is the broadest covered structure: graphs (consisting of nodes and connecting edges), or even more complex ones like hypergraphs?
- Are directed and/or indirected edges possible?
- Are multiple edges and/or loops supported?

5.1.2 Visual Representations

Apart from the structural relations that the graph library supports, also the possible representations of nodes and edges are of course an important criterion, because reaching the aim of supporting flexible modeling languages can be hindered by

environments that severely limit possible visual representations. In terms of section 4.1, this criterion can be formulated as related to the visual attributes that a library supports.

Apart from these general questions of flexibility in the representations, also the ability of a framework to manage multiple representations of the same conceptual entity is an important feature that can be of use in learning contexts in general and also within collaborative modeling in particular (Ainsworth, 1999; Löhner et al., 2003).

Another topic related to graph representations deals with spatial layouts: there are a number of modeling languages with have an abstract graph oriented representation (like, e.g., Petri Nets), but no inherent specification that determines the positioning of nodes. As such, automatic layout mechanisms embedded in the graph framework would be an advantage.

5.1.3 Syntax

If representations for elements of the graphs can be customized at all (cf. previous subsection), an interesting criterion is *how* these specifications can be done syntactically.

Of course, syntactical issues about specification formats might also relate to concepts exceeding pure representational parameters: if, e.g., semantic or operational features are available (cf. next subsection), also the specification methods for these are important.

For the purposes within this thesis, an easy but powerful mechanism of defining object and relationship characteristics and structures would be ideal. Yet, as known also from other fields of computer science, a certain trade-off between expressiveness and simplicity can be expected.

5.1.4 Semantics and Operational Functionality

In the context of expressiveness of graph structures and their specifications, also the potential of a framework to express semantics of nodes and edges is an important criterion - within the Reference Frame approach, the notion of semantics has a prominent place (cf. section 4.3).

The exact definition of model semantics varies considerably in literature (Harel & Rumpe, 2004), from the original meaning of a formal mapping between between syntax and a domain to aspects like, e.g., model behavior, context, or interpretation by humans. The tool comparison along the criterion of semantics will focus on the narrower original sense, but include other connotations if applicable.

Apart from the abstract notion of model semantics, another important criterion is the degree to which a graph framework can express *active* structures in the sense that model simulations are supported. This can, e.g, be reached through the provision of an event propagation mechanism, a general message passing framework, or states and state transitions for nodes and edges. Typical effects of actions on models may be related to the graph structure itself, or the semantics of the contained nodes and edges.

Obviously, an at least rudimentary support for operations on graph structures is a necessary requirement for a graph framework used in the context of modeling and simulation applications.

5.1.5 Interactive Usage

As pointed out in the introduction chapter, the implementations within this thesis are, among others, driven by the idea of allowing for an interactive and collaborative

system usage. For the used graph framework, this means that an interactive usage must principally be possible - pure display tools that, e.g., read graph definitions from a file and generate a visualization that cannot be further manipulated, are useless as underlying technologies. A subcriterion in the field of interactive usage is the potential of a library to restrict the structures that can be created by the user interactively: in many graph based modeling languages, not all graph structures constitute syntactically correct expressions (e.g., in a Petri Net, places and places must not be connected) so that a flexible constraint mechanism embedded into the graph framework would be an advantage. This criterion, however, does not claim that syntactical correctness should generally have the priority over freedom of actions - yet, the *simulation* of models will usually only be possible for syntactic correct ones.

A good fulfilment of the "interactive usage" criterion would be reached if, in addition to a general support for user interaction with the graph, also direct manipulation techniques like drag&drop were inherently facilitated by the graph library, as the immediate feedback (which the direct manipulation techniques offer) may be important for experimenting with models.

5.1.6 Additional Features

As will be shown in the next section, most graph libraries are designed for specific purposes or application areas. Therefore, they typically offer specific additional functions. A lot of these (like specific storage mechanisms, synchronization options, embedded advanced graph algorithms, or extraordinary performance in terms of memory or speed) are relevant for applications in the domain of collaborative modeling. For that reason, the inclusion of a dedicated criterion that allows for taking into account additional features makes sense.

5.2 Systems for Graph Based Representations

Having identified theoretical results about graphs and their representations, in particular also within the specific use case of modeling, this section lists existing *technology* in this field, and classifies it according to the criteria elaborated in section 5.1.

Nearly all modern programming languages come with an extensive library to support programmers in building applications with graphical user interfaces. Current examples include the Swing API of Java (Java Foundation Classes Homepage, n.d.), or the GDI+ tools of C# (C# Corner, n.d.). In addition to these built-in libraries, also an enormous number of other frameworks designed to facilitate the construction of graphical structures is available (Big Faceless Java Graph Library, n.d.; Java Imaging and Graphics Library, n.d.).

However, in contrast to this large number of frameworks that are targeted towards general *graphical* representations, specific tools for *graph* representations are rare - despite these representations being used frequently (cf. section 3.2). It seems that many applications use their own underlying technical infrastructure for the representation and visualization of graphs, but that these libraries are rarely encapsulated and offered as self-contained software components. The following subsections give an overview on the currently existing and freely available state-of-art graph libraries. Tools that offer graph *visualization* only and do not offer *interaction* options (in the sense of direct manipulation of graph structures, cf. subsection 5.1.5) to the user, like e.g. the dot framework (Gansner & North, 2000) or the Java Universal Network/Graph library (JUNG manual, n.d.), are not considered, as for the purposes within this thesis, interactive functions are a necessary prerequisite.

5.2.1 Sourceforge JGraph

A prominent Java library for graph representations in Java is the SOURCEFORGE JGRAPH project available at sourceforge.net and in a commercial version (Java Open Source Graph Visualization Component Suite, n.d.). This project provides a freely available and fully Swing (Java Foundation Classes Homepage, n.d.) compliant implementation of a *graph* component. The developments in the library are guided by the principles of full standards compliance, a clear and efficient design, and the avoidance of large or complex functions.

The SOURCEFORGE JGRAPH library is focused on graph *visualization* and offers a very high degree of flexibility in this respect. Some aspects of this flexibility are gained by specific functions of the library (like, e.g., zooming or grouping of elements), others are a direct consequence of design choices in the system architecture.

SOURCEFORGE JGRAPH is a modular system that makes use of a three-tier architecture. The lowest layer is core graph visualization library, the medium layer offers layouts, graph algorithms and export routines to various data formats, and the top layer contains applications which allow users to interact with the graph visualizations.

Most of the criteria listed in the previous section correspond to the lowest layer of this architecture. This layer makes use of the Model View Controller (MVC) software architecture pattern (Buschmann et al., 1996) in a similar way the standard Java Swing components do. This pattern allows the separation of data from its representation and contains an elegant way of change propagation. It is the de facto standard for current interactive systems with graphical representations.

In the case of SOURCEFORGE JGRAPH, the roles of the different parts are as follows (Alder, 2003):

- The *model* part (interface `GraphModel`) describes the abstract underlying graph structure, including groupings and selection models.
- The *view* part (class `GraphView`) contains the display's internal representation of the graph, and the mapping and update between the model and the view. A special focus is set on geometric relations and patterns.
- The *control* components (class `GraphUI`) handle the rendering process, the steps necessary for in-place editing and cell handling, and the objects involved in data transfer and marquee selection.

The class diagram in figure 5.1 illustrates the core parts of the architecture and visualizes the mentioned separation between model, view and controller. It is a remarkable aspect of the SOURCEFORGE JGRAPH architecture that this clear separation is only made on the *graph* level. In particular, there are no further substructures for nodes or edges. This has an immediate consequence on the expressiveness of the library: multiple views per graph model are possible and enable, e.g., two windows showing different parts of one graph, but this flexibility is not available on the node or edge level. Here, only one fixed visual representation at a time is supported.

An interesting detail in the architecture is that all elements of graph (models) are essentially of one type, the interface `GraphCell`. This interface allows for a very generic handling of attributes throughout the system. Apart from nodes (base class `DefaultGraphCell`) and edges (`DefaultEdge`), the SOURCEFORGE JGRAPH library makes use of so-called *ports* (`DefaultPort`) which are dedicated locations of nodes which serve as connection points for edges. This clearly illustrates the orientation of the library as a means of *visualization*.

There are two different ways of defining custom node or edge types in the SOURCEFORGE JGRAPH framework:

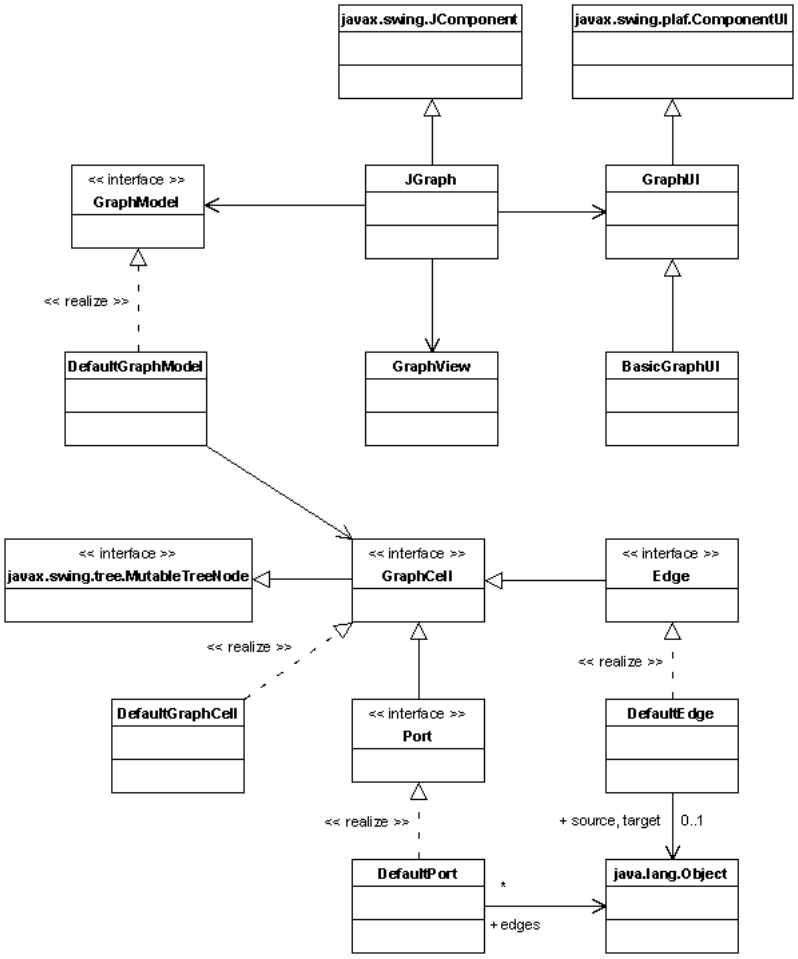

Figure 5.1: Architecture of the Sourceforge JGraph

- All `GraphCell` objects have attributes. These name/value pairs can be freely chosen and therefore serve as a (limited) means of creating custom element types.
- Java classes which implement the corresponding interface (`GraphCell` or `Edge`) or extend existing node or edge classes. This ensures a maximum of flexibility, but is of course no real option for non-programmers.

There is no formal *semantic* mapping associated to nodes, edges, or graph structures in the SOURCEFORGE JGRAPH library. However, it can be argued that the availability of attributes with open domains is a good base for embedding formal semantics. *Operational* aspects are well covered in the library: there are several event change propagation mechanisms following the Publisher Subscriber pattern (Buschmann et al., 1996) with respect to changes in models and views (not included in figure 5.1). The option of writing customized node and edge classes principally allows for a flexible reaction to these events.

Apart from pure visualization of graphs with flexible node and edge concepts, the SOURCEFORGE JGRAPH library also fulfills most of the criteria related to *interactive* usage: it allows for editing, moving, cloning, and sizing nodes, and bending edges as well as of course adding and removing edges and ports. The availability of each single form of interaction can be explicitly controlled.

The SOURCEFORGE JGRAPH library uses the XML based Graph Exchange Language GXL (Winter, Kullbach, & Riediger, 2002) for storage. This format supports typed, attributed, directed, ordered graphs (potentially also hierarchical structures and hypergraphs), and is designed as a means to reach syntactic interoperability between different applications that make use of graphs as central data type.

A very strong point of the library is its generic ability to support grouping of elements and hierarchic structures - a node can, e.g., contain a whole graph - the algorithms for deleting, selecting, and modifying elements fully take these relations into account. With the additional option of hiding specific parts of a graph, this enables applications to build quite complex structures with sophisticated navigation strategies.

Another aspect worth mentioning is the embedded capability of the library to undo and redo any kind of operation on the graph, also in the context of multiple views. This transaction based mechanism is implemented using the event propagation mechanism of the library. Even though the SOURCEFORGE JGRAPH framework does not have the ability to synchronize graphs between applications over a network, this could be a suitable starting point to add this feature - a structure that can react upon undo and redo events is already prepared to deal with remote change events to some extent.

5.2.2 TouchGraph

TOUCHGRAPH (Touchgraph, n.d.) is a library which is primarily designed for *visualization* of graph structures. It is very competitive in terms of performance: the management of several thousand nodes or edges is unproblematic, both in terms of memory usage and speed.

The main purpose of the TOUCHGRAPH library is put on assisting the user to visually navigate through dynamic networks. Current applications include a visual Wiki Browser, and a Google Graph Browser which uses the API of the search engine.

Unlike most of the other graph libraries presented in this section, the programming style within TOUCHGRAPH does not meet higher requirements: issues of flexibility and reusability are not dealt with, and some functions (like, e.g., the drag & drop support) are implemented in a proprietary way although current standard solutions do exist.

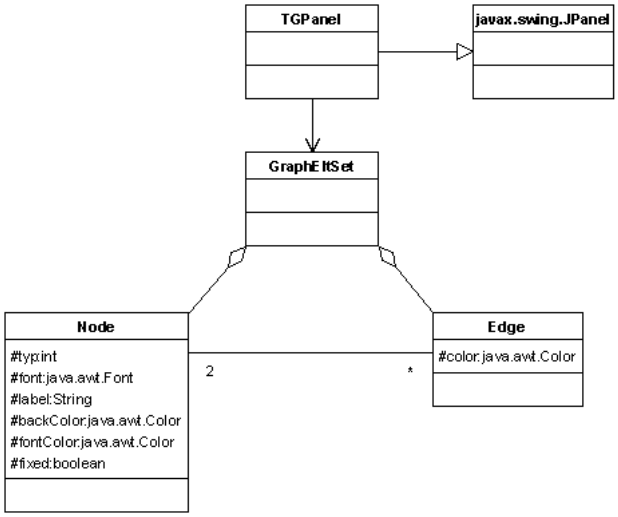

Figure 5.2: Architecture of the TouchGraph

The low degree of flexibility can be illustrated with the system architecture concerning node and edge management. Figure 5.2 reveals that there is neither a separation between models and views, nor any abstraction of node or edge interfaces. This implies that the definition of node or edge types beyond setting attribute values for color and displayed text (figure 5.2 contains all attributes for nodes and edges that are not directly related to spatial layout or unique identification) is hard to achieve. In fact, the library does not explicitly foresee a method to define custom objects, though the use of inheritance should work. Together with the absence of a suitable event model capable of propagating changes within a graph, this means that there is no real support for adding any kind of *semantics* (neither formal mapping nor of operational character) to TouchGraph instances

Another characteristic aspect of the TouchGraph library is that nodes and edges are no "real" Java user interface components in the sense of extending the `java.awt.Component` base class, but merely provide a paint method which adds their image to a `java.awt.Graphics` object. This has two immediate consequences. The first one is positive: this design choice is one of the reasons why the TouchGraph is fast and competitive in terms of memory usage. The negative aspects of this design include that it is not possible to make use of the flexible Java component library from within nodes in the TouchGraph. This means that there is no way of embedding *interactive* control elements like buttons in the user interface of a node.

Apart from the mentioned performance, strong aspects of the TouchGraph library are mostly related to visualization aspects: the tool generically supports zooming and rotation of graphs, and the partial hiding of subgraphs is well supported through the operations `expand`, `collapse`, and `hide`. Furthermore, the TouchGraph includes a sophisticated layout algorithm which bases on the principles that nodes have a repulsing effect on other nodes, while edges pull nodes. Making use of damping factors, the algorithm iteratively constructs a stable situation for any graph.

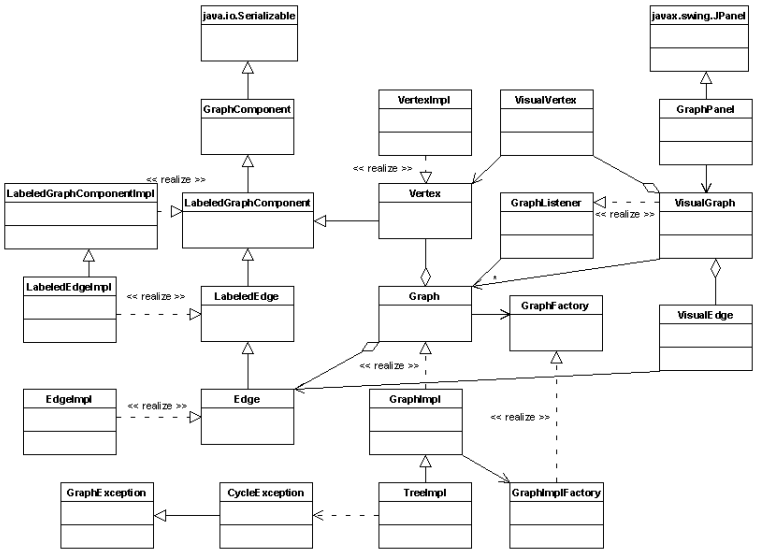

Figure 5.3: Architecture of the OpenJGraph

5.2.3 OpenJGraph

The OpenJGraph library (OpenJGraph, n.d.) is distributed via sourceforge.net. Apart from source code level documentation, it is not well documented. Similar to the TouchGraph, it does not seem to be maintained - however, both tools are compatible with current Java versions, which justifies their inclusion in this comparison.

OpenJGraph is a very flexible tool for graph *visualization* and *interactions* with graphs. The architecture of the library, as shown in figure 5.3, incorporates a lot of classic design patterns and bases on a clear separation between data and view. OpenJGraph does not offer a full MVC architecture, but the change propagation between model and view(s) is implemented with a Publisher Subscriber pattern, as in the original architecture.

A significant difference between OpenJGraph and both Sourceforge JGraph and COLLIDE JGraph (cf. next subsection) is that the OpenJGraph library applies the separation between model and view both on the graph level and on the node/edge level. This theoretically allows for multiple synchronous views per graph, and also per node or edge. In practice however, the tool does not allow for multiple views per node - yet, this could be overcome relatively easy by extending the visual components of the library.

An important part of the OpenJGraph library is an interface hierarchy (and corresponding implementing classes) to represent abstract graph structures. Directed and weighted edges (not included in figure 5.3) are generically supported, and adding elements (nodes or edges) to a graph is solved in a very elegant and flexible manner through a factory based solution (Gamma et al., 1995).

The OpenJGraph library has one critical design decision in common with the TouchGraph: the views for nodes and edges are no Java (Swing or AWT) components, but do merely consist of the provision of painting methods. As already stated in the previous subsection, this brings advantages in terms of performance, but severely restricts the design space for graph components, and in particular the options for interactive usage.

Apart from this disadvantage, the OpenJGraph tool has a lot of very strong

points: it includes a number of "standard" graph algorithms like the check for connectivity, the search for an eulerian tour, or the construction of a spanning tree. These algorithms are implemented very flexibly and with respect to potential reuse. Cycle checks, e.g., are implemented by using the Visitor pattern (Gamma et al., 1995), which elegantly allows to apply them also for extended node and edge concepts.

The OpenJGraph tool allows the specification of nodes and edges in XGMML, an XML format that describes graph structures and their visual attributes in detail. However, this format does not allow for customized fields, which might be interesting for the enhancement of the library towards the inclusion of semantics.

Operational semantics in the sense of allowing for active dynamic structures are partially supported in the OpenJGraph. There is the option to attach listeners to an abstract graph structure, but the events are only related to structural changes and (partially) to changes in the layout. There is no explicit consideration of the change of a model as a source of an event - in addition, the conception of nodes and edges as non-interactive objects does not easily allow for creating events of these types.

The layout problem for abstract graphs that have to be visually represented is solved very elegantly in the OpenJGraph library: via a Strategy pattern (Gamma et al., 1995), different layout managers (not to be confused with the standard Java layout managers) can be defined and applied to graph representations.

Finally, another aspect that distinguishes this library from the others in this comparison is the consideration of rules that *constrain* the set of "allowed" graphs. Any attempt to add nodes or edges is handled by the graph and may cause an exception to be thrown, which means that this action is not permitted. Thus, the definition of arbitrary rules is possible using inheritance (i.e. extending the graph class). Figure 5.3 shows how this approach is implemented in the library in order to construct a tree structure.

In general, the chosen method for representing constraints for graph structures is very flexible and expressive. Yet, a disadvantage of the approach is that rules are always bound to a graph. There is no explicit concept of a rule, which could be transferred between graphs, and there is no way of dynamically changing rules for a given graph.

5.2.4 COLLIDE JGraph

This last library within this comparison is the COLLIDE JGraph (COLLIDE JGraph, n.d.) which is being developed by the COLLIDE research group at the University of Duisburg-Essen. The design of this system is guided by the aim of high flexibility and expressiveness for the node and edge elements.

Similar to the Sourceforge JGraph, the COLLIDE JGraph makes use of the Model View Controller paradigm. However, the core principle of this pattern, the separation between data, control and representation, is implemented at a different point of the architecture (see figure 5.4). Each node and edge defines an explicit model and a view, but the overall structure - the graph itself - is not further subdivided.

This fact prevents some of the benefits of a pure MVC architecture: the COLLIDE JGraph library does not directly allow for having different graph views simultaneously. Also on the node and edge level, the COLLIDE JGraph has certain limits: although a separation into model, view, and controller is available here, the framework requires a one-to-one relation between models and controllers on the class and the instance level. Additionally, each controller can only handle only one view at a time, so that multiple simultaneous views on nodes are not possible.

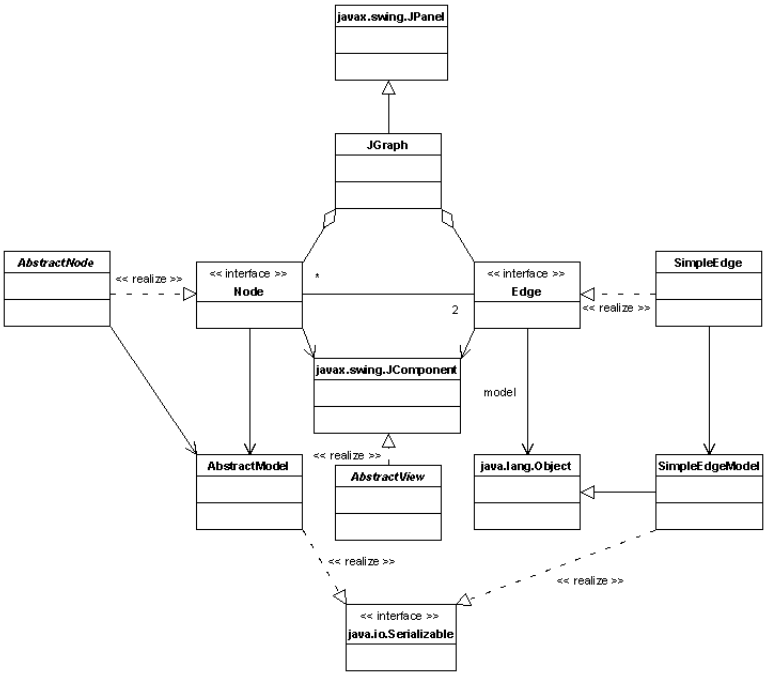

Figure 5.4: Architecture of the Collide JGraph

These restrictions obviously limit the original flexibility of the Model View Controller paradigm. The reason why these limitations are necessary is related to one of the advantages of the COLLIDE JGraph: graph instances can be synchronized between applications, on one machine as well as in a networked situation. This is implemented using the MatchMaker technology (cf. 5.4.7) the following way:

- To synchronize itself, the COLLIDE JGraph creates a MatchMaker synchronization tree that consists of one single vertex.
- For each node in the COLLIDE JGraph, a child is added to the root vertex of the tree. This child contains information about the location of the node in the COLLIDE JGraph. Below this vertex, another child vertex is added to the tree. The content of the latter child is the model of the node in the graph. This structure of the subtree decouples location changes from model changes, which allows for very fast transmission of position changes of nodes (e.g., during drag&drop operations) without the need to send the whole node model each time.
- For each edge in the COLLIDE JGraph, a child is added to the root element of the MatchMaker tree. The content of this child vertex is the model of the corresponding edge in the graph. As the location of the edge is implicitly determined by the location of the nodes that it connects, no split similar to the one for the case of the nodes is done here.
- The local change of a model in the COLLIDE JGraph is propagated to synchronized instances via the MatchMaker library. The remote COLLIDE JGraph instances are notified using the event mechanism of the MatchMaker library and react upon transmitted changes by updating their local models correspondingly.

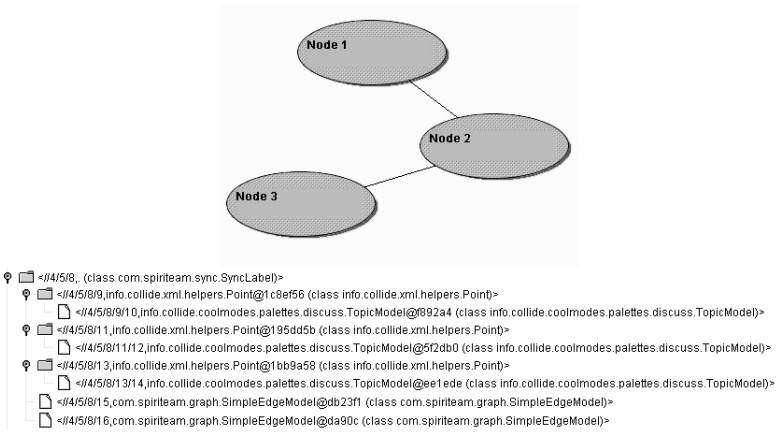

Figure 5.5: A COLLIDE JGraph instance and the corresponding MatchMaker synchronization tree

- Node or edge creation and deletion events are distributed in the same way.

Figure 5.5 shows a typical synchronization tree used to couple two COLLIDE JGraph instances via MatchMaker. The three `TopicModel` instances describe the nodes contained in the graph and consequently contain the string content of the nodes - more sophisticated nodes can of course contain more attributes. The two `SimpleEdgeModel` instances are related to the two edges in the graph. Edge models always contain identifiers of the two nodes that the edge connects, together with additional information of the edge (e.g., a label or a color).

The approach chosen to couple COLLIDE JGraph instances is very flexible and lightweight: the node and edge models that build the foundation for synchronization are usually very small data classes with only few attributes. Due to the chosen event propagation mechanism (Publisher Subscriber pattern), unnecessary data transfers are prevented. Additionally, the usual case of a change in the graph structure (e.g., caused by the editing of a node, or the creation of an edge) does not affect the whole synchronization tree but only one leaf, so that the amount of data that needs to be transmitted is further reduced.

Another dimension of flexibility, which is a result of the design choice of model based synchronization in combination with the MVC paradigm, is that a model can principally be displayed differently on different synchronized applications, as each graph instance decides locally and individually how to represent a model.

The disadvantage that comes with the flexibility of the chosen method for synchronization is that the whole approach relies on the node and edge models only as the primary means of synchronization. A first consequence of this is that they need to be suitable for serialization. This is usually unproblematic and no real limitation for model classes in the MVC architecture. Apart from this, another restriction is that in the chosen method of synchronization, the application (i.e. the COLLIDE JGraph) must be able to receive node models and generate "complete" nodes including view and controller components from the model information - for the edges, the same situation occurs. This is the reason for the necessity of the one-to-one relation between models on the one side, and controllers / views on the other side.

One partial exception to this is that multiple views per model are indeed possible in three senses:

1. It is possible to switch between different views easily. In this case, only one

view at a time is active so that the 1:1 relation holds. The information which view is currently active can be stored in the node model.

2. Locally, different synchronous "views" for a model can be implemented by simply generating user interface components within the controller class, and displaying them on the screen. Yet, these "additional views" would then be differently bound to the graph: whereas the "first" view is embedded in the core interfaces (and thus generically accessible in the library), the additional ones would be proprietary and outside the scope of the COLLIDE JGraph. As a consequence, synchronization features for these views (and many other functions like, e.g., drag&drop support) would have to be added manually.

3. Multiple synchronous views are also possible using the synchronization mechanism. By not including information about the currently "active" view in the node model (or simply ignoring is), different graphs can contain different views of the same model.

Despite these two options, there is no generic support for multiple and synchronously available node (or edge) views in the COLLIDE JGraph. Additionally, adding such a support is a difficult task since the role of the models in the architecture is very central: not only the synchronization support, but also the whole storage mechanism, and the unique identification function of nodes relies on the models.

Apart from the mentioned architectural differences and the available synchronization support, the COLLIDE JGraph is similar to the Sourceforge JGraph in many respects: there is a suitable *event* mechanism to distribute model changes to subscribed listeners, and an *interactive* usage is fully supported, including drag & drop operations.

The *syntactical* definition of graph elements in the COLLIDE JGraph architecture can be done either in an intuitive (but proprietary) XML storage format, or in Java. The first method relies on existing Java classes that offer the option to be parameterized with specific attribute values of primitive data types, the latter one essentially makes use of inheritance and allows for a very flexible way of specification - but, as in the case of the Sourceforge JGraph, only for users with programming experience.

Also the degree to which *semantics* of graphs is covered by the COLLIDE JGraph is similar to the Sourceforge JGraph case: there is no embedded semantic mapping, but the option of extending the predefined nodes and edges, and in particular their models, is a suitable starting point. The built-in event propagation mechanism is a part of the library that is well suited to include constraints for graph structures - however, this is not embedded in the core COLLIDE JGraph tool.

Finally, the COLLIDE JGraph contains interfaces for attaching automatic graph layout engines. One of these engines, which makes use of the same algorithm as in the TouchGraph tool, is also included in the library.

5.2.5 Comparison and Discussion

The previous subsections described the characteristics of the current state-of-art libraries for graph representations. Table 5.1 summarizes these descriptions with respect to the criteria list as presented in section 5.1. The assessment symbols used in the table express the degree to which the libraries fulfil the criteria. They have the following meanings:

\+ Completely fulfilled

Table 5.1: Comparison of graph libraries

	Sourceforge JGraph	**Touch-Graph**	**Open-JGraph**	**COLLIDE JGraph**
Structures	graphs	graphs	graphs	graphs
Extensions	ports, nested graphs	nested graphs	-	-
Directed Edges	+	+	+	+
Multiple Edges	∘	∘	+	(+)
Loops	∘	∘	+	(+)
Flexible Representations	(+)	-	-	(+)
Multiple Representations	(+)	-	(+)	-
Layout Engines	+	(+)	+	(+)
Syntax Specification	XML, Classes	Attributes	XML, Classes	XML, Classes
Semantic Mapping	∘	-	∘	∘
Operational Functions	+	-	(+)	+
Interactive Use	+	(+)	(+)	+
Constraints	-	-	+	∘
Drag & Drop	+	(+)	(+)	+
Storage	+	-	+	+
Synchronization	-	-	-	+
Graph Algorithms	+	-	+	(+)
Other	zoom, undo, hiding	zoom, rotate, hiding	-	-

(+) Partially fulfilled
∘ Basically prepared but not elaborated
- Not fulfilled

The summary table immediately reveals two things: there is no single graph library that fulfils all the criteria, and a simple determination of a library which is "the best" according to the criteria list is not possible, as each of the frameworks has different strong aspects but also some weak points.

Common points are that all compared suites use graphs (rather than hypergraphs or other more complex structures) as their abstract data structure, and that directed edges are always supported. The syntactical definition of nodes and edges is mostly done in XML, which usually means that the values of a more or less predefined set of attributes can be specified. All libraries fall short in the provision of semantics: none of them easily allows for the inclusion of any kind of semantic mapping. The only reasonable way of adding this (and other functions needed for modeling with the graph structures) is the definition of node and edge classes in Java and their embedment in the tool. Using inheritance mechanisms, this is principally possible in all the libraries. Yet, the degree of support for this varies considerably: TouchGraph offers no support at all, Collide JGraph and Sourceforge JGraph provide suitable interfaces and guidance how to implement custom structures, and OpenJGraph even includes a number of helpful pattern based mechanisms that allow for a finer integration of custom node and edge types into its graphs.

A distinctive aspect between the libraries is the way in which they solve the task of providing visual representations for nodes and edges. Two different princi-

ples are observable: either elements are painted in a graphics context, or the Java component API is used. The former approach, which is taken by TouchGraph and OpenJGraph, is a severe restriction for the purposes within this thesis, as it drastically limits the potential of interactivity for supported node and edge types: things beyond simple low level mouse clicks, like e.g. control elements (checkboxes, buttons,...) or even simple text input fields within elements, are not easily possible.

This drawback reduces the usability of the TouchGraph and the OpenJGraph as a base technology for the further implementations in this thesis. In the case of OpenJGraph, this is despite the fact that this library has by far the most flexible architecture and also some very elegant solutions for detail features (e.g., the layout and algorithm implementation, or the consideration of rules).

The remaining two libraries, Sourceforge JGraph and Collide JGraph, present themselves very differently: the former is excellently documented, well supported by a large user community, and distributed in a commercial spin-off, whereas the latter is not published well. Yet, with respect to most of the comparison criteria, these two libraries are very similar. The most important point, from which a number of other similarities can be derived, is that both make use of the Model View Controller paradigm (in different parts of the architecture), and reach a certain degree of representational flexibility with this.

The differences between the two tools are rooted in their origin and primary purpose: the Sourceforge JGraph is designed as a graph visualization tool, which explains its strength in the areas of layout and display. The advantage of the Collide JGraph, a tool developed to support collaborative usage of graphs, is its function of synchronization.

In summary, my conclusion is that both the Sourceforge JGraph and the Collide JGraph are suitable as a base for implementing a collaborative modeling system based on the Reference Frame approach and its visual typed graphs. I have chosen to use the Collide JGraph in the implementation parts because the already available synchronization support seems to be more valuable for the targeted implementations than a finer control of visual parameters, which a choice of the Sourceforge JGraph would have offered.

5.3 Criteria for Synchronous Cooperation Support

Aiming at the implementation of a framework which allows the co-construction of visual typed graphs, a considerable technological ingredient is a system for cooperation support, especially the support for synchronous cooperation in distributed scenarios. As the focus of this thesis is not to *build* a technical solution for cooperation support but to *use* one, the aim of this section is to set up a criteria list which then serves as a means for the comparison of several communication frameworks in section 5.4.

Literature on distributed systems (Coulouris, Dollimore, & Kindberg, 2000; Tanenbaum & Steen, 2002) and groupware (Borghoff & Schlichter, 1998) helps to identify these distinction criteria between different communication technologies and frameworks. The following subsections briefly describe the typical criteria that can be used to characterize "lower level" communication frameworks. There are definitely also other important topics to discuss in the area of supporting cooperation via synchronized applications. These will shortly be presented in subsection 5.3.6. Yet, these typically depend on the concrete synchronized application - the system comparison in section 5.4 will rely only on criteria which depend solely on the underlying communication technology.

5.3.1 System Architecture

The architecture of a distributed system is essentially dominated by the location and roles of its components and the types of relations between them (Coulouris et al., 2000). Typical architectures for distributed systems include the following ones:

Pure Client-Server. This frequently chosen and most important architecture consists of one dedicated server process and a number of client processes that establish connections with the server and interact with it.

Modified Client-Server. A lot of variants of the pure client-server model can be found. Most of them try to improve the central weakness of the original architecture, the bottleneck of the central server. Typical modifications are the inclusion of multiple servers, or the use of proxy and cache components.

Peer-to-Peer. A peer-to-peer architecture strictly avoids explicit server processes and completely relies on direct communication between the peer clients. The client processes have similar roles in this architecture - sometimes, even the same application is used on all client machines.

Heterogeneous Systems. Some architectures are mixtures of the two extremes of client-server and peer-to-peer architectures, and differ in such a way from both extremes that they cannot be called modifications any more. Typical examples of heterogeneous systems include the extension of a peer-to-peer architecture with specific dedicated processes for, e.g., central resource management. Other heterogeneous architectures rely on a classic server, but additionally allow for explicit inter-client communication.

Other means of distinction between different architectures include the availability of replicated data, of mobile code or mobile agents (with code and data), and of explicit layers with dedicated functions.

Obviously, these architectural dimensions of a communication framework have an impact on the applications that use the framework. For instance, the availability of a central server can simplify certain tasks (e.g., logging), whereas a pure peer-to-peer solution avoids bottlenecks and the need to install dedicated servers.

5.3.2 Model of Information Transfer

The model of information transfer describes lower level aspects of data transmission between processes. As such, it contains information about the following points:

- Is data transmitted directly from client to client, or are indirect channels used? Are specific multicasting, broadcasting or routing techniques made use of?
- The criterion of speed: what is the mean transmission time for messages, and which factors have an impact on the performance?
- In how far are security issues addressed? Are, e.g., encryption mechanisms embedded, and how can users identify themselves?

Of course, the identified criteria are not independent and have an impact on each other. Indirect data transmission, e.g., is generally problematic in terms of security, and a higher performance can be gained by omitting data integrity checks which reduces the overall reliability. Additionally, some of the criteria may not be directly applicable to certain communication techniques. It often makes sense to

handle specific issues (like, e.g., certain security aspects) on a generic low layer, which can then be used by various higher-level frameworks.

Despite these two things (interdependency and applicability), the inclusion of an information transfer model as described above in the criteria list makes sense, as it covers the most important technical constraints for application communication. Evidently, these are a considerable factor for the choice of a communication technology that underlies the targeted application.

5.3.3 Strategy for Information Distribution

The strategy for information distribution as summarized by Borghoff and Schlichter (1998) is on a higher level than the more transmission oriented model of information transfer. In particular in at least partially replicated architectures, the following criteria are an important characteristic property of communication technologies.

- Is there a general *policy* of information distribution in the system? Typical examples for such policies are event based change propagation mechanisms, or pure client-server models which require pull strategies.

- Are there any basic *conceptual entities* of interaction between processes? Typical primitives are messages, streams, remote objects, and remote method invocation (Tanenbaum & Steen, 2002). Another dimension relates to the usage of these primitives: in how far is synchronous or asynchronous communication enabled?

- Does the communication technology impose any *prerequisites* on the data that is supposed to be sent?

It is worth noting that not every technology does necessarily have to address all these aspects. In fact, most low-level fundamental technologies are likely to be neutral with respect to some criteria. However, the strategy for information distribution in terms of the listed criteria has direct implications for the applications which rely on these strategies. In particular, certain communication features on the application level can be significantly supported by means of a well-chosen underlying infrastructure.

5.3.4 Stability

Also on a higher level than data transmission reliability and failure detection (which are included in the "information transfer model" criterion), the stability of a communication technology is an important characteristic point. In distributed settings, the following aspects are worth consideration:

- Are there any (lower or higher order) fault tolerance mechanisms?

- Is persistency guaranteed by some means?

Often, the answers to these questions are to a large extent predetermined by other criteria like the architecture or the model of information transfer. Yet, as sophisticated higher order mechanisms may make up for certain lower level disadvantages, an explicit category for stability related issues makes sense.

5.3.5 Concurrency Control

When users are interacting and access or modify shared data, conflicting situations can inevitably occur. If a communication technology offers a means to control this concurrency, this is an important characteristic factor, as each solution to some extent predetermines application behavior. For instance, locking based pessimistic concurrency control strategies may require the application to "freeze" and block user input, whereas optimistic techniques can potentially lead to inconsistent states and may require undesired undos of user actions in order to re-establish consistency.

The following two questions determine the internal and external aspects of concurrency handling in a communication framework:

- Are there critical aspects that require concurrency control even *within* the framework? By which means are these controlled? These technology inherent concurrency problems can, e.g., occur in replicated architectures.

- What other means to deal with general concurrency problems that inevitably occur when several users synchronously operate on shared entities does the technology offer? How good are these means in terms of data consistency and respond times?

5.3.6 Higher Level Criteria

As already outlined in the beginning of this section, there are also a number of higher level criteria that determine the quality and usefulness of groupware tools in general, and in particular also collaborative systems. These depend on the application rather than on the underlying communication mechanism, so that it does not make sense to include them in the criteria list for the system comparison in section 5.4. Yet, to underline the importance of higher level aspects, also as a reference point for the system description parts of this thesis, the following short list provides a brief summary of some important points.

Awareness

When using groupware systems, it is often important for a user to know about the context of his and other people's system usage activities. This type of knowledge is denoted with the term *awareness*. Endsley (1995) gives a very intuitive and easy to understand term definition of awareness as

> *"[...] knowing what is going on."* (page 36)

This definition implies that different types of awareness can be distinguished, including e.g. situation awareness, conversational awareness or social awareness.

Gutwin and Greenberg (2004) analyze the importance of workspace awareness, which they define as the knowledge of a user about other user's interaction with a shared workspace. The collaborative situations they refer to are characterized by a real-time distributed groupware that relies on shared workspaces and is used by small groups. This matches quite well the target within this thesis, so that their main message is of importance:

> *"For people to sustain effective team cognition when working over a shared visual workspace, our groupware systems must give team members a sense of workspace awareness."* (page 177)

Coupling Level and Mode

Some important architectural design choices related to the communication framework have not been mentioned in subsection 5.3.1 because they depend on the architecture of the application that is used cooperatively.

For the important case of a Model View Controller architecture (Buschmann et al., 1996), both Schümmer and Schuckmann (2001) and Suthers (2001) discuss variants of synchronization that, e.g., consider shared and private states, different object of synchronization (model, view, or controller), and strict versus loose coupling strategies.

User Management

Building collaborative software in most cases involves the need of having some kind of system side distinction between different users. A basic exemplary use case that requires this distinction is an *authentication* as one part of a security mechanism (cf. subsection 5.3.2). Apart from these aspects that can indeed be implemented by a very low level communication framework, there are also a number of higher order functions which substantially contribute to the usefulness of groupware tools and require user management.

The availability of user *identifications* within the system is a prerequisite for a lot of awareness mechanisms, as it allows for mirroring back to users not only, e.g., changes caused remotely, but also the causing factor of these changes. Beyond that, user *models* build the foundation for a number of applications in collaborative systems - they can, e.g., be used to propose potential collaboration partners (Hoppe, 1995; Ikeda et al., 1997).

Group Interface

An important characteristic aspect of collaborative systems is the interface that the system offers to the user group. The most important aim of a group interface is to provide the users with shared context (Borghoff & Schlichter, 1998), the concept of awareness as discussed before representing one aspect of the group interface. Other aspects include the choice of the core means of interaction (like, e.g, a shared whiteboard), or the principle that underlies the conceptual sharing method. An example of the latter is the well-known WYSIWIS (what you see is what I see) approach.

Transparency

Transparency of a distributed system is the hiding of its heterogeneity, the fact that processes and resources are physically distributed. A number of different transparency types like, e.g., access, location, or concurrency transparency are distinguished. In general, a transparent distributed system presents itself to the users as if it was "just" a single computer system (Tanenbaum & Steen, 2002).

Transparent groupware systems have the advantage of a natural usage: the cooperative usage should not be more difficult than a single user mode. Yet, it is worth noting that the aspects of awareness and transparency are in some senses contradicting to each other, so that the resulting application profile (what is made transparent / what is made explicit) is an interesting characteristic factor for a groupware tool.

Feedthrough

As stated in section 1.2, communication through the artifact can be an important means to support collaboration. The mechanisms of making shared objects reflect the manipulations done on them are called feedthrough mechanisms (Dix et al., 2004). Of course, these mechanisms are application specific and cannot be completely implemented (but supported) by underlying communication technologies.

5.4 Technical Solutions for Cooperation Support

Trying to give a complete overview about existing technologies that are used to support cooperation in distributed systems is unrealistic: the amount of distributed systems targeted at this aim is far too high. Yet, while some of the systems use their own proprietary approaches to implement the information transfer between computers, others rely on existing and reusable technologies or even frameworks of higher complexity.

The following subsections compare a selection of existing technologies and frameworks with respect to the criteria list elaborated in section 5.3. The selection of technologies has been motivated by the premise of considering only frameworks that are independent of specific applications. To further reduce the set of communication frameworks to be reviewed, only those that are freely available, currently maintained, and independent of at least either the operating system or the programming language are considered. Pure application sharing tools which essentially mirror application views like, e.g., NetMeeting (NetMeeting Resource Kit, n.d.) are not included in the list because they do not offer enough degrees of flexibility for sharing applications: using them, an application is either completely shared or it is not - some important implementations within this thesis will, however, require a finer granularity. Another argument against view based sharing mechanisms is that they do not allow for real data replication: as only a view is synchronized, a crash of the machine which really runs the application causes all others to definitely loose their data.

Taking into account that the communication technology, despite being a considerable external resource used within this thesis, is far less important than the library for graph representations, the communication technology review is kept shorter than the comparison for the graph libraries as conducted in section 5.2. In the following descriptions, I just mention the basic approach of the respective frameworks, and describe their performance in terms of the criteria.

The following subsections are organized as follows: the review starts with approaches that are independent of both both operating system and programming language. This category (subsections 5.4.1 - 5.4.3) contains very basic techniques that simply reach the independency because they are very low level, as well as very sophisticated frameworks that put a lot of effort in reaching this degree of independency. In subsections 5.4.4 to 5.4.7, some tools that depend on the programming language but not on the operating system are outlined. Finally, some important examples for approaches that are independent of programming languages but dependent on the operating system are presented (subsections 5.4.8 and 5.4.9).

5.4.1 Network Sockets

Network sockets are a very basic form of establishing communication between computers in a network: all modern operating systems support networking via sockets, and all the currently available general purpose programming languages allow for the the development of programs that make use of network sockets.

The idea behind sockets is simple: the processes that communicate use network ports to send data to each other. The transmission can either be connection-oriented using, e.g., TCP (in this case, data streams are the means of communications), or connectionless using e.g. UDP. In the latter case, the data units are called datagrams. TCP ensures the arrival of all packets and takes care of ordering issues, whereas UDP datagrams may get lost without any notification to sender or receiver.

As mentioned, sockets are independent of both operating system and programming language (of course, concrete data bytes sent via the sockets might have interpretations that depend on either of the two). Sockets are by far the fastest way of communication between two machines, but (being very low-level) offer no embedded functions for concurrency control, or persistency. The Secure Socket Layer (SSL) standard, supported by major programming languages, is a widely accepted means to encrypt socket based communication. Via multicast and broadcast mechanisms, the mode of information transfer in socket based communication is more flexible than a simple one-to-one model.

5.4.2 Web Services

Web services (Coyle, 2002) are a modern middleware approach that can be understood as a consequent enhancement of remote procedure calls towards independency of operating system and programming language. The basic idea of web services is that SOAP (Simple Object Access Protocol) envelopes containing XML data are transported via standard web based transmission technologies (like, e.g., HTTP).

Web services architectures are based on explicit servers and clients. Service description and location is possible via the WSDL (Web Services Description Language) and the UDDI (Universal Description, Discovery and Integration) protocols. Due to the use of established standard technologies, the encryption of messages is easily possible (e.g., using HTTPS), and there are also higher-order means of fault tolerance (e.g., using SMTP). Concurrency and persistency issues, however, are not generically handled by web services.

5.4.3 CORBA

The CORBA (Common Object Request Broker Architecture) approach (CORBA specification, n.d.) has been developed by the Object Management Group, an industry consortium, in the 1990s. It is a broker architecture which reaches independency of both programming languages and operating systems through the use of the textual Interface Definition Language (IDL), which is mapped to language-specific constructs. Distributed objects, as specified by their IDL interface, make up the core of CORBA. ORB (object request broker) services are running on all participating machines (clients and servers), which makes the architecture principally usable in peer-to-peer situations, though normally CORBA is used with dedicated clients and servers.

The information transfer is done directly between machines (though there are interface and implementation repositories that are used for locating services), and as one of the few architectures within this comparison, CORBA supports both the pull and the push model for information distribution. Through this and the available event and notification mechanisms, CORBA offers very flexible methods of interaction between distributed objects, including asynchronous method calls.

Via services (implemented as system services described in IDL), CORBA allows for concurrency control with transactions, higher order fault tolerance mechanisms (e.g., replication), persistency, and a number of security and authentication facilities.

5.4.4 Remote Method Invocation

The use of remote objects to invoke methods on them, a successor of Remote Procedure Calls (RPCs), is (besides network sockets) the second basic networking technology that is available in most modern object oriented programming languages. A remote procedure call is typically dependent on the programming language - Java RMI is at least independent of the underlying operating system, which justifies its consideration in this comparison.

RMI always involves exactly two machines with different roles: one acting as a server and the other one as a client. As one application can act as both server and client, RMI is suitable as a base technology for peer-to-peer architectures - but, of course, also for client/server based applications. RMI calls are relatively fast (though not reaching sockets level performance) and insecure in their basic implementation, though the exchange of the transport socket layer is possible and allows for the inclusion of encryption mechanisms. The primitive data type transmittable via RMI are Java objects that implement a dedicated tagging interface (`java.io.Serializable`) which indicates the ability to be transformed into a byte stream.

Being among the very basic technologies for programming distributed systems, RMI does not provide any means for enhanced fault tolerance, persistency, or concurrency control.

5.4.5 Java Shared Data Toolkit

The Shared Data Toolkit for Java Technology (JSDT) (Java Shared Data Toolkit Homepage, n.d.) is a library that is designed to allow programmers to easily develop collaborative applications. JSDT relies on sessions, which are defined as a set of channels that can transport data objects (which essentially consist of byte arrays) to groups of applications.

There is one dedicated server application which manages a registry - the whole communication among the client applications has full-duplex multicast characteristics in the sense that data can be sent to and received from the channel. Of course, also "private" communication between two applications via a channel is possible.

JSDT offers event subscription and notification mechanisms, with events fired whenever clients enter or leave channels, or when data is transmitted.

JSDT allows the choice between three different underlying transportation techniques: sockets (TCP/UDP), HTTP (to easily tunnel firewalls), and LRMP (Lightweight Reliable Multicast Protocol). Concurrency can be controlled by a token-based distributed synchronization mechanism, which can be used to ensure mutually exclusive access to a resource. Apart from the reliability mechanisms of TCP, JSDT has no further fault correction mechanisms, and no provision of persistency. The latter would even be in contrast with the concept of a channel as a medium to transport volatile information.

5.4.6 JavaSpaces

JavaSpaces is a technology which belongs to the Java Jini suite (Freeman, Hupfer, & Arnold, 1999; Flenner, 2001). The central idea behind JavaSpaces is that of a shared whiteboard in form of a tuple space with associative lookups (Carriero & Gelernter, 1989): objects in the space have a type, and a set of attributes with name and value. A lookup in a JavaSpace is done by first creating an element of the desired type, then setting values for all the attributes that serve as queries, and finally querying the space with this template. The JavaSpace then returns an

object that matches the given type and attributes - there is no embedded way of accessing objects through any kind of direct identifier.

As described, JavaSpaces are a classic client/server architecture. JavaSpaces does not offer a way for clients to directly communicate with each other beyond the exchange of objects via the space. The currently available implementations are quite slow and offer only very limited ways of secure access. However, there are commercial initiatives (GigaSpaces Homepage, n.d.) that aim at solving these issues.

JavaSpaces can be accessed by four basic operations: `write`, `read`, `take`, and `notify`. The first three access and modify the content of the shared whiteboard, the latter gives access to the event mechanism embedded in JavaSpaces: an application can get a callback when an object which matches a query object is put into the space.

As mentioned, the data primitive used in JavaSpaces is an object. In contrast to other technologies that can handle all objects which offer the basically needed function of serialization, JavaSpaces adds some other (minor) constraints like, e.g., the availability of public variables or a no-arg constructor. The reasons for these is the specific serialization mechanism used for the lookup strategy. The constraints do not really restrict the generality of the approach, but may mean a lot of work if existing applications are to be synchronized via JavaSpaces.

JavaSpaces (at least the commercial implementations) offer persistency services, but no advanced fault tolerance mechanisms. Concurrency control is solved by a pessimistic locking approach with transactions.

5.4.7 MatchMaker

The MatchMaker library (Jansen, Pinkwart, & Tewissen, 2001) for synchronization of Java applications has been developed at the University of Duisburg-Essen. MatchMaker mixes characteristic elements of centralized architectures (e.g., a dedicated server) with others that are typically found in peer-to-peer networks: applications are not strongly affected by network problems, and even a split from the server does not necessarily cause a MatchMaker client to crash.

Applications that use MatchMaker for synchronization have no way of directly communicating with each other. Similar to JavaSpaces, MatchMaker foresees only indirect communication via shared objects. In the case of MatchMaker, these shared objects build a tree and thus have a certain structure. Figure 5.5 illustrates how this tree structure can be used to represent the structure of the artefact that is shared by means of MatchMaker.

It is worth mentioning that in a MatchMaker setting, all client applications usually have local replicas of the MatchMaker tree and do not have to access the data stored on the server all the time (though the library would in principle allow for this way of access).

Due to the indirect communication (all changes caused by clients are first sent to the server, which then propagates them) and the underlying RMI technology, MatchMaker is relatively slow. This is a trade-off for the advantage of easily allowing late-comers to receive the complete state of the synchronization tree, and storing the tree even if no client applications are currently connected to the server.

As mentioned, the data primitive of MatchMaker is a tree. Each node in the tree is composed of two things: a unique identification label which allows for object identity (in opposite to the approach chosen by, e.g., JavaSpaces), and a content object. The latter can be any java object which supports serialization. Thus, MatchMaker can handle exactly the same objects as the underlying layer (RMI) and does not add further constraints.

Like some other libraries in this comparison, MatchMaker has an event notification mechanism that bases on a Publisher Subscriber model (Gamma et al., 1995). MatchMaker distinguishes between three types of modifications within a synchronization tree:

- modifications of the tree structure itself by adding or removing elements,
- state changes of objects within the tree, and
- the execution of actions on elements of the tree

According to the publisher subscriber model, objects in a client application can register themselves as listeners for elements of the tree. They then receive notifications about events occurring on that tree element, and can modify their local state correspondingly.

Together, the event mechanism and the data structure of MatchMaker allow for a very fine granular control of coupling between applications. Typically, the objects synchronized via MatchMaker are of the same type in all client applications and are registered as listeners of their corresponding element of the synchronization tree. Both, however, is not a necessity: MatchMaker easily supports, e.g., the synchronization of a checkbox in one application with a toggle button in another application - which makes sense, as the conceptual models underlying these two objects are similar. MatchMaker also allows an object in one client application to register as listener for arbitrary elements in the synchronization tree. This facilitates, e.g., the development of additional services using the framework (see below).

As pointed out, a strong point of the MatchMaker event model is its flexibility. Also the support of both state based and action based propagation mechanisms offers a lot of design options to the user of the MatchMaker library. In some cases, the use of actions to distribute local changes to coupled applications is the most suitable, elegant and/or efficient way. Examples include the communication of a "button clicked" event, or situations where the state of an object is large and changes are frequent - here, the transmission of changes (encapsulated in actions) is advantageous. Other applications may choose a purely state based synchronization (which seems the more natural usage of MatchMaker, since a tree of object states is the core data structure in the library), and decide not to use actions at all.

One disadvantage of the allowed mixture between action based and state based methods is that this makes logging and analysis of log data far more difficult, as most analysis methods rely on either actions or states, but seldom support a mix between the two. This, of course, is a drawback especially in research contexts that focus on interaction analysis. Another disadvantage of the implementation of actions in MatchMaker is that actions can cause the state of the synchronized object to change without generating further state change events transmitted to the client applications. This is necessary if the aim is to reduce the amount of transmitted data by only sending changes, encapsulated into actions. As for the mentioned reasons also the MatchMaker server locally keeps one instance of the synchronization tree, this requires that also on the server, the actions are executed on the corresponding elements of the synchronization tree. Consequently, a MatchMaker server can only synchronize classes that are available in the server runtime environment, and cannot deal with "unknown" data types. This definitely is a restriction in a number of usage scenarios.

Security issues are not directly addressed in MatchMaker, thus this library does not exceed the options already provided by the underlying RMI layer. As in the case of CORBA, replication can be seen as a fault tolerance mechanism. In fact, a strong point of MatchMaker is that it allows coupled applications to continue working (locally) even if the communication server fails.

Within a master thesis (Jansen, 2003), MatchMaker has been extended with persistent storage functions (reached by regularly backups of the server replica), and a central redo/undo mechanism which relies on the event model and essentially transforms and reorders events. Also an approach for pessimistic locking concurrency control by means of transactions has been implemented for MatchMaker within a master thesis (Mwakitalima, 2003).

5.4.8 DCOM

Within this comparison, DCOM (DCOM Technical Overview, n.d.) is the first example of a library that is independent of programming languages but dependent on the operating system. DCOM is the extension of the Microsoft Windows component object model (COM) towards usage in distributed systems. Though criticism on the design of the library is not rare, DCOM has at least definitely proven to be useful in a lot of real world scenarios (Tanenbaum & Steen, 2002).

The general architecture of DCOM resembles that of CORBA: client and server machines have local service control managers (whose functions are similar to the ORB in CORBA), and there is a central interface repository. Thus, as in the CORBA case, both client/server applications and peer-to-peer scenarios are possible with DCOM.

Like, e.g., RMI and CORBA, DCOM makes use of a remote object model. Yet, a central drawback of DCOM is its lack of a central naming and location service. The Microsoft Active Directory technology can be used for these purposes, but if the latter is not available, no name based lookups are possible, which constitutes a severe limitation.

The basic notion within DCOM is that of a *component*, which is defined as an executable object that can be activated and that can interact with other components. Components are transient, i.e. they are destroyed when they are not referenced by other components. The interfaces of components are binary, which makes DCOM independent of programming languages, but dependent on platforms. Both interfaces and implementing components have a globally unique interface identifier.

Similar to CORBA, DCOM allows for synchronous and asynchronous method calls. In addition, DCOM offers the storage of events for temporarily not active consumers.

DCOM contains authorization mechanisms to control the access to remote objects, and the library uses underlying Microsoft Windows layers for encryption using SSL and Kerberos.

Concurrency and fault tolerance are addressed within DCOM by transactions with automatic backup. Also persistent data storage is possible (both for objects and also for events) using database access.

5.4.9 GLOBE

The GLOBE (GLobal Object-Based Environment) system (Steen, Homburg, & Tanenbaum, 1999), developed at the University of Amsterdam, has been designed primarily for distribution transparency and is driven by the principle of designing for scalability.

Compared to other higher level libraries as presented in the previous subsections, GLOBE can be characterized as a puristic architecture with only few central services for naming and location. GLOBE formally distinguishes between clients and servers, but this distinction is on a technical level. With respect to the design aims of the library, GLOBE is well suited for peer-to-peer applications.

In contrast to most other technologies (including DCOM, CORBA, and RMI from this comparison), GLOBE does not adopt the remote object model. Instead, it

uses a distributed shared object model, which bases on replication and distribution of objects and their use by multiple processes. In contrast to MatchMaker, GLOBE does not define a common policy for replication and distribution, however: this specification is done individually by each object. This approach offers enormous options to the programmers using GLOBE, but of course also means additional work when developing the objects to be shared.

Objects in GLOBE are a composite of essentially four components which encapsulate the *semantics*, the *connection* to the underlying network, the *replication and distribution* strategy, and a *control* component. The object model of GLOBE is passive in the sense that between method invocations, objects are not active. The only way of accessing objects in GLOBE are synchronous method calls.

As mentioned, GLOBE is a system with only few central services: most tasks (including policy definitions) are delegated to the distributed objects themselves. Consequently, security issues are addressed within GLOBE by adding encryption mechanisms on a per-object base - there are no global security services. Persistency and fault tolerance can be solved in GLOBE in an elegant manner using the sophisticated replication mechanism, but concurrency control is problematic. There are ways to define object-bound control mechanisms, but GLOBE does not provide inter-object synchronization mechanisms (Tanenbaum & Steen, 2002).

5.4.10 Comparison and Discussion

Even the brief description of the communication frameworks as done in the previous subsections reveals that there is a wide variety of approaches and solutions with respect to the comparison criteria mentioned in section 5.3. The presented technologies differ greatly in terms of purposes, underlying philosophies, and design principles. Furthermore, they partly use each other and thus belong to different layers in an abstraction hierarchy.

The degree of fulfilment of the comparison criteria varies considerably, and there is no simple scale with which the different communication technologies could reasonably be compared on a +/- base as done with the graph libraries in section 5.2.5. I therefore take a different approach and compare the presented libraries more in terms of the intended usage (i.e., supporting the sharing of visual typed graphs within an implementation of the Reference Frame approach) than on a general level.

The first important point is that within this thesis, the communication framework will be used to synchronize graph structures. Of course, the *data primitive* of the communication framework should match these purposes. This practically disqualifies both the direct use of sockets and also JSDT: both approaches are oriented towards data packets or streams, and do not support well the object oriented usage context.

The second critical requirement is a consequence of the targeted collaborative scenario: users are expected to work with representations, and changes are supposed to be communicated to collaborators. Here, the availability of *events and notifying mechanisms* in the underlying communication technology of essential importance. Web services, RMI, and GLOBE do not offer generic event support and are therefore not well suited for the purposes of this thesis.

The "remaining" four technologies are CORBA, JavaSpaces, MatchMaker, and DCOM. The next important requirement is related to the intended flexibility and openness of the collaborative modeling system: a flexible and expressive *object model* that does not restrict the usage scenarios is an advantage here. The stateless transient object model without global identifies of DCOM does not meet these requirements. The tuple space approach together with the object model of JavaSpaces is also critical with respect to this point: there is no easy way of impos-

ing a structure onto elements within a JavaSpace, and the missing object identity and lacking options of directly accessing elements within a JavaSpace are serious drawbacks.

As a conclusion, I consider both CORBA and MatchMaker as suitable communication technologies for the implementations within this thesis. CORBA requires a lot of implementation overhead, but is better supported and documented, programming language independent, and contains more flexible solutions for a lot of detail problems. Consequently, it would be the first choice in a selection without side constraints. However, as also MatchMaker fulfils all the needed essential requirements and, in addition, is already supported by the chosen graph library (cf. section 5.2.5), it is reasonable to use MatchMaker as communication technology.

Chapter 6

An Abstract Implementation Model

The Reference Frame approach as presented in chapter 4 is on a conceptual and formal level. Having discussed suitable "background" technologies in the previous chapter, this chapter now proposes an implementation of the Reference Frame approach based on these technologies. For several reasons, it would be inappropriate to design the targeted collaborative modeling framework directly on top of the results from chapter 4 and the libraries selected in chapter 5:

- Though the conceptual results have been developed with a view towards the construction of a concrete system, the Reference Frame approach is not operational enough to provide guidance for a straightforward implementation.
- The attempt of implementing the results from chapter 4 directly is likely to lead to mixing decisions deduced from the theoretical foundations with more HCI or usability related aspects, which have to be taken into account for the concrete system development, but not necessarily on an abstract architecture level.
- It is an established and reasonable tradition in software engineering to design for flexibility and reusability. In particular, as will be shown in chapter 7, an encapsulated abstract implementation of the Reference Frame based modeling approach decouples conceptual work from system design issues and thereby allows differently targeted or designed tools to make use of the Reference Frame idea.

For these reasons, the presentation of an implementation of the Reference Frame approach is done first on an intermediate layer of abstraction within this chapter of the thesis. In particular, the conceptual results from chapter 4 are taken up, and abstract computational structures that represent a basic functional implementation of these conceptual results are developed, using an object oriented approach.

6.1 Visual Typed Graphs

The notions of typed graphs and their layouts have been presented in chapter 4 on an abstract level. This section, discusses how they can be implemented using the COLLIDE JGraph (COLLIDE JGraph, n.d.) library, which is described in detail in subsection 5.2.4 within the comparison of graph libraries.

Any attempt of building a system for visual typed graphs has to give an answer to two design questions: the representation of node and edge types, and the implementation of layouts and visual attributes. These are discussed in the following subsections.

6.1.1 Node and Edge Types

In object oriented approaches, a natural way of representing the *types* of certain entities is to use different classes corresponding to the types. The interfaces and abstract base classes provided by the COLLIDE JGraph library are well suited for this. Here, any class that implements the interface `info.collide.graph.Node` can be conceived as a node type, and in analogy all the classes which implement the interface `info.collide.graph.Edge` represent edge types (cf. figure 5.4).

With this translation of the type concept into an object oriented structure, the node type set $\mathcal{N}$ and the edge type set $\mathcal{E}$ from definition 4.3 naturally correspond to the sets of classes which implement the interfaces mentioned above.

Node and edge types defined in terms of classes, the nodes and edges themselves are represented as objects - instances of these classes. Therefore, the "instance of" relation is exactly the implementation of the type mapping relations dom_N and dom_E as contained in definition 4.3.

These simple relations show that the COLLIDE JGraph library (like most of the libraries compared in section 5.2) offers a straightforward way of representing typed graphs. There are even several options the library offers which are not required by the notion of typed graphs:

- The attributes and operations of node and edge classes do not have any correspondence in the basic concept of typed graphs.
- Inheritance mechanisms allow the definition of nodes as being subtypes of other nodes (and similar for the edges). In the concept of visual typed graphs, this is represented through the subtype relations in definition 4.2 - yet, these are not required (though supported) by the core parts of the theory.

The following parts of this chapter describe how these additional capabilities can be used to support the intended usage of the typed graphs within a collaborative modeling framework.

6.1.2 Layouts

As presented in chapter 4, the layout information for graphs is kept separate from the concepts of types and Reference Frames: the attribute sets V_N and V_E are independent. This design decision was made in order to easily allow for a generic integrated visualization of a typed graph, irrelevant of the node and edge types that this graph contains.

On the software level, this can be modeled by (interface or implementation) inheritance: abstract node or edge structures which define (among others) the visual attributes, and methods for interfacing to the general top-level rendering methods of the `info.collide.graph.JGraph` class. In terms of definition 4.4, the domains of the variables represent the attributes themselves, and the results of the mappings λ_N and λ_E are represented through the values that these variables have for a given set of nodes and edges.

Table 6.1 shows the visual attributes of two simple exemplary base classes in the COLLIDE JGraph library. Here, the layout parameters of the node contain information about its current selection state, its variability in size, its behavior concerning layout algorithms, and its popup menu. In addition, a `javax.swing.JComponent`

Table 6.1: Visual attributes of `AbstractNode` and `SimpleEdge`

Class	Visual attributes

`AbstractNode`	`boolean selection`
	`boolean resizable`
	`boolean sticky`
	`JPopupMenu popupMenu`
	`JComponent view`

`SimpleEdge`	`EdgeComponent[] edgeComponents` `JComponent view`

encapsulates further pieces of information (including position, border, etc.). The parameters of the edge are its components (visual objects to attach to the edge when painting it), and again a `javax.swing.JComponent`.

Higher-level base classes with more visual attributes do also exist in the library (e.g., the classes `AbstractView` and `ShapedEdge` in the `info.collide.graph` package). This solution of handling the visual attributes in abstract base classes and interfaces enables a type-independent availability of visual information, which mirrors the approach taken in definition 4.4.

One drawback of this approach is that arbitrary mixing of layouts and types is not possible: at design time of a class, a fixed set of layout attributes (i.e., the base class and the implemented interfaces) must be chosen. Here, more dynamic mechanisms (e.g., variable property lists) could help - however, up to now, no real need for this occurred during our practical usage of the system.

As briefly described in this section, the COLLIDE JGraph library enables an easy and straightforward implementation of visual typed graphs, with classes representing types, and variables in abstract base classes or interfaces for nodes and edges corresponding to visual attributes.

6.2 Rules as Expressions for Constraints

Apart from the pure availability of visual typed graphs as suitable data structures for a collaborative modeling system, also the notion of syntactical correctness of models is important.

In section 4.2, the syntactical correctness of models was addressed through *constraint mappings* - a visual typed graph being correct (or: conform), if the constraint function parameterized with the graph evaluates to true.

Definition 4.6 does not require a specific calculus for the definition of constraint mappings. Thus, its implementation is a design choice, influenced by two parameters: on the one hand, the mechanism should be intuitive to use, as it is one part of the definition of Reference Frames, which should not be too complicated (cf. requirements list in section 1.5). On the other hand, a constraint mechanism with a too small scope does not make sense, as it risks not supporting modeling languages with complex syntactical requirements would not be supported any more.

Taking these two positions into account, one possible solution is to base the implementation of constraint mappings on *rules* which are easy to specify and expressive enough to cover a broad range of possible syntax restrictions, but do not offer any further dynamic calculus. A rule merely represents a certain non-allowed situation in a graph, the calculation whether the rule matches is left to the graph.

Obviously, rules of different complexity are imaginable. This can be taken into account by different rule *types*, which enable the expression of different non-allowed

situations in visual typed graphs. In particular, the following rule types can be distinguished (and have been added to the COLLIDE JGraph library):

Edge Rules. This very basic type of rule can be used to express that between two nodes of a specific type, only a certain number of edges (rule weight) of a specific type are allowed. Setting the rule weight to 0 results in simply forbidding connections of the node types with the edges types. These simple edge rules can, e.g., be used to disallow the connection of places with places in Petri Nets.

Limit based Edge Rules. This rule type allows, in addition to the parameters of edge rules, the specification of limits for the *total* number of outgoing and incoming edges per node, depending on edge type and neighbor node type. Limit based edge rules can be used to control on a fine-granular level the data flow in a graph.

Cycle Rules. These rules are parameterized with a list of edge types and a list of node types. The rule matches if the graph contains a circle that consists only of nodes and edges of the specified types. Cycle rules can be used to guarantee tree structures, e.g., in the case of calculation trees (cf. figure 4.2).

Structural Pattern Rules. This very generic type of rule is parameterized with a typed graph, serving as a pattern. The rule matches if the graph to be checked has the pattern as a subgraph: in the matching algorithm, node and edge types are considered. Pattern rules are very flexible, as they allow the specification of very complex forbidden "axioms" - their limit is that they are not dynamic (i.e., only fixed patterns are allowed, see discussion of rules with behavior below).

Equality based Pattern Rules. This rule is very similar to the structural pattern rule, differing in that the matching check does not rely on the type of nodes and edges, but invokes the `equals()` method of the elements. This gives greater flexibility in pattern matching by delegating decisions to the node and edge classes.

These five types are partially redundant: e.g., a pattern rule can easily replace a simple edge rule by using a graph that consists of two nodes and one edge as a pattern. This redundancy has two reasons. First, these rule types have evolved during a number of years (practically, a new rule type was added whenever it was noticed that the available ones were not sufficient any more), so that the less powerful constructs still exist for *compatibility* reasons in the software. Second, the feedback from the programmers indicated that in the case of several suitable rule types, the easier one is chosen. This motivates keeping the redundant types for *convenience* reasons.

Alternative constraint mechanisms with more expressive power might support the specification of expressions with certain behavior, e.g., specified by a graph grammar (cf. subsection 2.2.2). This would obviously extend the set of constraints that can be expressed, but would at the same time make the specification of a rule more difficult.

Technically, these rules are implemented as Java classes (extending a common superclass `info.collide.graph.Rule`, which encapsulates localized messages that are designed as feedback for the user who "made the mistake"). The `JGraph` class has methods for managing these rules, and is also responsible for checking whether these rules match. In theory, this problem breaks down to the subgraph isomorphism problem, which is computationally hard (cf. subsection 2.1.2). Section 6.4 shows an implementation that addresses these problems.

6.3 Reference Frame Interfaces and Implementations

Similar to the visual typed graphs, the concept of Reference Frames has been introduced in chapter 4 on a formal level, which was helpful for conceptualization purposes, and in particular for the derivation of some advanced mappings and relations. This section now proposes an object oriented implementation of Reference Frames. In contrast to the case of the visual typed graphs, where the implementation was a mapping to the existing COLLIDE JGraph library, the object oriented Reference Frame implementation is not based upon any pre-existing component: it merely uses the visual typed graphs implementation in the COLLIDE JGraph as underlying data structure.

6.3.1 Encapsulated Components

As stated in definition 4.10, a Reference Frame is a conceptual collection of the following components:

- Node and edge types
- Visual attributes for nodes and edges
- A set of constraint mappings
- A semantic domain and a semantic mapping
- A synchronization context mapping

Given that Reference Frames are the analogy of modeling languages in my approach, a high degree of flexibility in their specification is of high importance. Therefore, it makes sense to model a Reference Frame as a Java interface, in order to allow the specification of modeling languages independent of inheritance hierarchies - in the case of a programming language with multiple inheritance, also an abstract base class would of course have been a reasonable choice. Figure 6.1 shows a list of all the methods in the interface.

The first block of methods in the interface is designed to support the dynamic and interactive use of Reference Frames in the system. In particular, a Reference Frame can define a user interface, and has methods that allows a runtime system to notify it about changes. Details about this will be described in the next chapter of this thesis.

The second block serves practical purposes. Here, a Reference Frame specifies a unique identifier, several pieces of metadata which are usable for managing multiple Reference Frames, and some methods that allow for a smooth embedding in the runtime system (cf. chapter 7).

Node and Edge Types

The two methods that return `Association` arrays allow the specification of node and edge types. Here, an `Association` is a collection of two classes: a model class and a controller class (in the MVC scheme). This way of specification is necessary due to the JGraph and MatchMaker libraries: both need to generate views and controllers from models (cf. sections 5.4.7 and 5.2.4), so that an explicit specification of models *and* corresponding classes (to be generated from these models by Factories in the runtime environment) is needed.

In terms of the conceptual approach from chapter 4, the proposed Reference Frame implementation explicitly specifies the node and edge types that belong to his

```
public interface ReferenceFrame {
public void init(GraphApplication manager);
public void update();
public void dispose();
public Palette getPalette();

public String getIdentifier();
public LocalizedMessage getLocalizedIdentifier();
public String getLocation();
public String[] getPackagePrefixes();
public String[] getNeededReferenceFrames();
public String[] getNeededExternalResources();
public String[] getAuthors();
public String[] getLanguages();
public ActionType[] getActionTypes();
public void switchMode (int mode);

public Association[] getNodeAssociations();
public Association[] getEdgeAssociations();
public Node[] getImportedNodes();
public Edge[] getImportedEdges();
public Rule[] getRules();
public void synchronizeContext(Node node, JGraph graph);
}
```

Figure 6.1: The technical ReferenceFrame interface

DEFINES set (using the `getNodeAssociations` and `getEdgeAssociations` methods). This implementation allows for the following:

Proposition 6.1 *The proposed way of defining node and edge types and Reference Frames allows for guaranteeing unambiguity of Reference Frame sets.*

Proof. First, the name of a class uniquely identifies this class in a Java runtime environment, so that the mapping between (conceptual) node or edge types and their representing classes is really bijective.

Now assume that a set R of Reference Frames is given. In the proposed implementation, each Reference Frame $\mathcal{R} \in R$ declares his DEFINES set through interface methods. Therefore, an application that manages the set R just has to check the returned lists for elements contained in more than one, and can thus easily detect violations of the unambiguity criterion.

A practical implementation can then exclude one of the Reference Frames that caused the problem from the set R - in the Cool Modes system (cf. next chapter), the consistency check is, e.g., performed actually *before* adding a Reference Frame to the set of already "loaded" ones, which ensures permanent unambiguity of the set of loaded Reference Frames.

Similar to the DEFINES set, there are methods in the Reference Frame interface which allow the direct specification of the KNOWS set. Theoretically, the node and edge classes that a Reference Frame refers to on the code level, including those returned in the `getAssociations` methods, could constitute its KNOWS set also implicitly. None of the further implementations presented in this thesis indeed rely on the explicit way of defining the KNOWS set: e.g., the model interpretation mechanism works by delegating the calculation of the $G_{|\mathcal{R}}$ graph to the Reference

Frame. Yet, the explicit way of declaring imports has the advantage of clearly reflecting the relationships between Reference Frames.

In addition, if a Reference Frame relies on the availability of functionality provided by other sources (e.g., if certain needed algorithms are implemented in external Reference Frames), it is possible to declare this dependency in the `getNeededReferenceFrames();` and `getNeededExternalResources();` methods of the `ReferenceFrame` interface, returning the identifiers of the needed Reference Frames or the location of the general resources. This indicates the dependency between the Reference Frames to the framework, which can then check the availability of the resources at runtime.

Constraints

As described in detail in section 6.2, the implementation of constraint mappings is done by rules. Via the `getRules()` method, a Reference Frame can explicitly specify a list of syntactical integrity constraints. Subsection 6.4.1 discusses the management of the rule sets by the framework.

Visual Attributes

The visual attributes for nodes and edges (V_N and V_E in the formal definition) are defined implicitly: as stated in section 6.1, these are contained in the abstract base classes and interfaces for the node and edge structures defined by the Reference Frame. Being a basic means for consistent visual representation of heterogenous models, the visual attributes are not intended to be highly dynamic. Yet, adding attributes if possible through the inclusion of further interfaces.

Semantics

As contained in section 4.3 and definition 4.10, the semantics to be encapsulated within a Reference Frame consists of two elements: a semantic domain D and a semantic mapping *Ip*, which maps to D. The "standard" domain of *Ip* are graphs - though more modular approaches which allow the mapping of nodes or edges to D are also possible (cf. section 4.3).

Similar to some other cases (like the the visual attributes), an implicit way of representing semantics in the implementation, driven by the following principles, makes sense:

- As described, nodes, edges, and Reference Frames are represented as classes which implement specific interfaces or inherit from certain abstract base classes. Apart from the requirement of being suitable for serialization (which comes from the collaborative context of use), there are no further explicit or implicit restrictions concerning attributes or operations of the classes.
- Therefore, the semantics of a graph (or: node/edge) can be stored in attributes of the nodes or edges classes, or in attributes of the Reference Frame class. This is a natural approach, as some attributes will typically be related to single elements of a model, whereas others describe the model as a whole (difference between $Ip(n_G)$ and $Ip(G)$). This way, the semantic *domain* can be conceived as the cross product of the domains of these variables.
- In accordance with the previous point, the semantic *mapping* can be represented by methods which calculate the values for the variables that constitute the semantic domain. These methods can be specified either in the node or edge classes, or in the Reference Frame class (here, of course, delegations are possible).

Alternative design decisions foreseeing, e.g., a central storage for semantics attributes, are possible with the abstract Reference Frame concept. An advantage of a more centralized approach is that it simplifies the access to the result of model interpretation significantly - however, it imposes (typically) unnecessary conditions on the developer of a Reference Frame implementation. Besides the choice between centralized and distributed storage of semantic attributes, another design decision is related to whether an explicit declaration (or: marking) mechanism for semantic attributes is used or not. The next part of this section and section 6.4 show that such a mechanism is not required for the core functionality needed for the interpretation of distributed heterogeneous models. Thus, to relieve the developer from unnecessary work, the proposed architecture does not foresee an explicit marking mechanism. Subsection 8.3.3 shows that some users of the system libraries support this design decision, whereas others would have preferred a more explicit approach.

Synchronization contexts

The synchronization context mapping has an explicit representation in the `ReferenceFrame` interface: the method `synchronizeContext(Node, JGraph)` accepts a node and a visual typed graph as parameters. In conformance with definition 4.8, the policy for implementations of this method is that it calculates a synchronization context of the parameter node in the parameter graph (cf. proposition 4.2 - in the best case, a *minimal* synchronization context should be returned), and couples the whole context together (instead of only the node). This way, an attempt of synchronizing a node in a graph with other graph instances can be processed *locally*, and lead to a coherently synchronized subgraph.

The inclusion of the `synchronizeContext(Node, JGraph)` method in the interface requires the developer of a modeling language to reflect in detail upon the semantics of the structures he defines in partially coupled situations. Of course, this is not a trivial task, especially not for modeling languages with formal semantics.

An alternative to this, which disburdens the developer from these decisions, would be the *automatic* calculation of (in the best case minimal) synchronization contexts. However, such a calculation is practically not reasonable due to the flexibility of calculating model semantics as stated above. In particular, the following three steps would be required:

1. a way to make explicit all the variables in nodes, edges, and the Reference Frame itself which belong to the semantics (i.e., to distinguish them from other information like, e.g., temporary variables or help variables),

2. a method to compare these variables with partially synchronized applications, and

3. a technique to build the minimal synchronization context based on the results of the comparison.

In particular the third point is the problematic one:

- Straightforward algorithms that simply test out which subgraph is a minimal synchronization context are no real option due to their complexity, especially taking into account that this is a distributed algorithm (step 2).

- The idea of relying on the comparison results (step 2 in the algorithm) does not work either: even if the nodes and edges with varying semantics are available, the step of determining which elements of the graph must minimally be synchronized in order to "repair" the inconsistency cannot be derived by simple means due to the possible complex interdependencies in the semantic

mapping and its a priori unknown calculation. The case is even harder if semantics information is (at least partially) stored in the `ReferenceFrame` implementation itself: if this varies between two instances, there is no direct way of finding out if and how this influences the semantics of single nodes, and thus no easy method of finding a non-trivial synchronization context. A known storage location (marking of semantics attributes) does not help here, and approaches without a central storage location for semantic attributes are generally problematic, as they have to use substitutes for general model attributes that are not bound to specific elements of the model.

- Finally, algorithms that go into detail about the interdependencies and are able to calculate a suitable synchronization context based on the comparison results are very similar (and not less complex) than the ones required for implementations of `synchronizeContext(Node, JGraph)`. Both approaches also offer the same degree of explicitness.

In order to assist the developer in the task of defining suitable synchronization contexts (which do not have to be minimal, though non-minimal ones lead to an unnecessary restriction of options for partial synchronization), a number of typical algorithms that are applicable for a variety of modeling languages can be pre-defined and implemented in form of a Strategy pattern (Gamma et al., 1995):

Single Node. This simple implementation does not add any nodes to be additionally synchronized. This algorithm makes sense if the semantics of a node does not depend on the surrounding context in the graph. A visual language designed to support brainstorming sessions with unrelated contributions is an example of a language where this strategy makes sense.

Whole Graph. The second trivial case always synchronizes the whole graph upon the attempt to synchronize one node. This strategy guarantees a synchronization context, but obviously in most cases synchronizes too many elements.

Connectivity Component. In modeling languages where the graph structure plays an important role, the semantics may often be retained if the connectivity component that a node belongs to is synchronized along with the node. Here, highly interoperable Reference Frames are an example (cf. the System Dynamics example in the next subsection).

Subgraph induced by Reference Frame. Typically, for "closed" (non-interoperable) modeling languages the subgraph of the graph which consists only of types known by the Reference Frame (denoted by $G_{|\mathcal{R}}$ in section 4.6.1) is a good candidate for a synchronization context. Here, calculation nets or the extended Reference Frame for Stochastics (Kuhn et al., 2004) as described in the next subsection are representative example languages.

The class diagram in figure 6.2 illustrates how these algorithms are implemented using the Strategy pattern. It contains the class `AbstractReferenceFrame`, which is an abstract adaptor class (in the sense of the Java adaptor classes) which contains default implementations for most of the methods contained in the ReferenceFrame interface.

6.3.2 Implementations for Interoperability

In the introduction of the Reference Frame concept in section 4.5, two important aspects were paid attention to: first, the *conceptual* structure of a Reference Frame representing a "modeling language", and second the *interoperability* options between

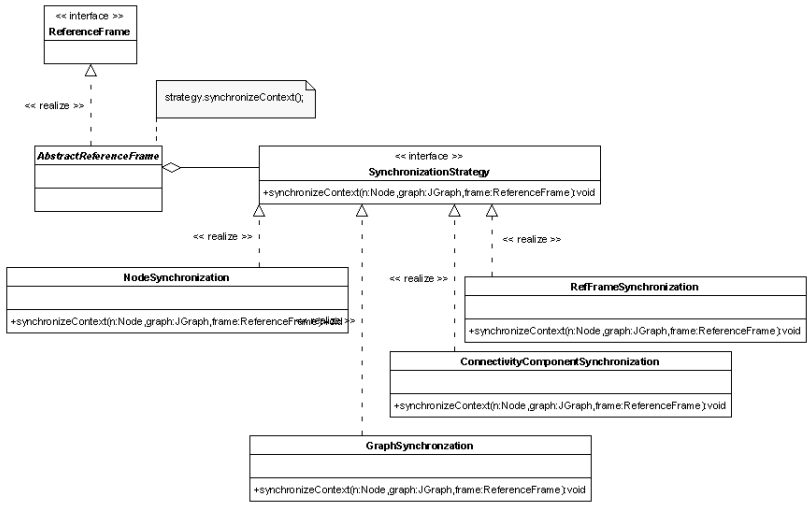

Figure 6.2: Synchronization contexts as strategies for Reference Frames

Reference Frames. There, two different ways have been proposed, one being import based, and the other one extension based. This subsection now outlines possible implementations of these two kinds of interoperability between Reference Frames. A number of other forms of interoperability, which does not strictly adhere to the "import" or "is-a" type, are imaginable and possible on the code level.

Import based solutions

Definition 4.12 can be read in the sense that an *import* relation between two Reference Frames $\mathcal{R}_1$ and $\mathcal{R}_2$ exists if $\mathcal{R}_1$ includes a set of node types $\mathcal{N}' \subseteq \mathcal{N}(\mathcal{R}_2)$ and/or a set of edge types $\mathcal{E}' \subseteq \mathcal{E}(\mathcal{R}_2)$ to his set of "interpretable" structures. In terms of section 4.6, this import $\mathcal{R}_1^{\swarrow\langle\mathcal{N}'_2,\mathcal{E}'_2\rangle}$ means that the KNOWS set of $\mathcal{R}_1$ includes $\mathcal{N}'_2$ and $\mathcal{E}'_2$.

A prerequisite for discussing import relations between Reference Frames is that the node and edge type import is well-defined (i.e., DEFINES and KNOWS sets do not share elements). Using the unambiguity property of Reference Frame sets, the propositions 4.3 and 6.1 show that this can be reached with the architecture proposed in this chapter.

Figure 6.3 shows an example of the import relation between Reference Frames. The left side and the right side of the figure show two Palettes, which are user interfaces of Reference Frames (cf. next chapter).

The left Palette belongs to a Reference Frame in the field of mathematics. It contains an element that allows the user to specify a function term, and another one that allows the input of number pairs in a table. A plotter component is capable of visualizing the information contained in elements of the other two types (function plots or single points from table). In addition, this Reference Frame imports a "simple edge" type and offers it to the user in the Palette. Here, the distinction between defined and imported elements is transparent: in the user interface, it cannot be seen. The user interface of a Reference Frame can be flexibly defined (cf. section 7.2), so that also other approaches are imaginable: in general, neither the defined nor the imported types have to be contained - however, at least the defined types will typically be.

The right Palette shows a "System Dynamics" Reference Frame (Bollen, Hoppe,

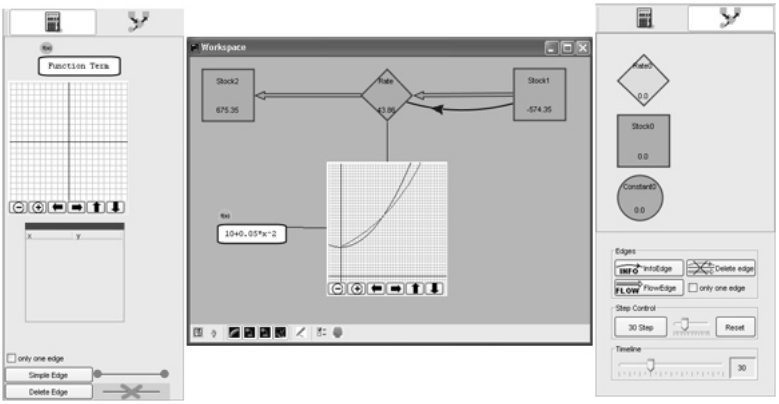

Figure 6.3: Node imports between Reference Frames

Milrad, & Pinkwart, 2002). This enables computational modeling using the methodology introduced by Forrester (1968). The Reference Frame defines three node types (stocks, rates, and constants), and two edge types which represent data flow and information flow. Though not shown in the Palette, this Reference Frame imports the plotter component of the other Reference Frame.

This import relation becomes clear in the workspace, shown in the center of the figure. The upper part contains a simple System Dynamics model, which describes an exponential growth. This part of the model is connected to the plotter component, which then shows both the graph of the function entered in the textual field on the left, and also the graph of the exponential growth. This kind of connection between objects defined in different Reference Frames is obviously more than a simple syntactical connection that an abstract graph structure can offer.

On the programming level, the import was in this specific case done by an interface that the "data source" nodes implement. The plotter node is capable of accessing the numerical data contained in the nodes which implement this interface. Thus, the import relation is done here by implementing this specific interface and thus making use of a subtype relation on the node type level. Yet, a number of other code-level equivalents of the conceptual "import" relation are imaginable.

Extension based solutions

As argued, import relations between Reference Frames can be implemented by simply using concepts (node or edge classes or interfaces) defined by other Reference Frames. The example described above outlines that this simple relation can already offer quite nice options.

Yet, some information like constraint mapping sets, semantics, or synchronization context mappings cannot be transferred between Reference Frames with the import mechanism proposed above. Thus, another technique for the implementation of the "is-a" relations is needed. Here, a natural way is the use of inheritance. If, on the code level, a specific Reference Frame class `c1` extends another Reference Frame class `c2` and does not override any method, it immediately fulfils some of the criteria listed in definition 4.13: the constraint mapping sets defined by `c1` and `c2` are the same, and the semantic domains also practically fulfil the criterion (with D_E consisting of one neutral "dummy" element).

The only problematic issue are the node and edge types (and, implicitly, the visual node and edge attributes). The node and edge associations returned by `c1` and `c2` are the same, which violates the unambiguity criterion. This exceptional

situation, however, can easily be detected by the application framework - in the case of Cool Modes (cf. next chapter), the Java Reflection mechanism is used to decide whether a "real" double definition occurs, or whether the double definition is caused by classes that are in an inheritance relation.

As argued, if `c2` is an implementation of the `ReferenceFrame` interface, and `c1` extends `c2` *without overriding any methods or adding attributes*, then the "is-a" relation in the sense of definition 4.13 holds. Not surprisingly, these trivial extensions are semantically consistent in the sense of definition 4.15, as neither the syntactic constraint mappings are modified nor the semantics mapping/domain are changed.

However, a more interesting question is of course if the "is-a" relation (or even the syntactic or semantic consistency) also holds if `c1` overrides methods of `c2`. Here, three cases have to be distinguished:

Types and visual attributes. If `c1` *imports* further node types (not defined or imported by `c2`), this leads to an extension of $\mathcal{N}'^*$ in the terminology used in definition 4.13. Obviously, this cannot violate the subset criterion. The case of `c1` *defining* new node and edge types is irrelevant within the criteria list in the definition. The inclusion/definition of edge types is analog to the case of the node types. Finally, the subset relation of the visual node/edge attributes is an immediate consequence of the subset relations for node and edge types together with the implicit definition of visual attributes (cf. subsection 6.1.2).

Semantic Domain. As stated, the semantic domain of a Reference Frame is conceived as the cross product of data types of the variables that constitute the semantic domain. In the proposed implementation, these variables can either be associated to node classes, edge classes, or the Reference Frame class itself. If `c1` extends `c2`, then the variables introduced additionally by `c1` (either in the class itself, or in additional node and edge classes), are the set which constitutes D_E in terms of definition 4.13. `c1` cannot reduce the semantic domain of `c2` due to the functionality of the inheritance mechanism in Java.

Constraints. This is the only critical case. If the `getRules()` implementation in `c1` does not include the rules already defined by `c2`, then the constraints criterion specified in definition 4.13 is violated.

The previous lines show that the way in which specializations of Reference Frames deal with syntax (i.e., which constraint mapping sets they define) is the factor that determines if an extension on the code level is really equivalent to an "is-a" relation in terms of definition 4.13. However, there are a number of techniques that can be applied on the framework level (and, thus, independent of concrete Reference Frames), which ensure certain characteristics of extensions. As these techniques are related to the runtime management of several Reference Frames, they are discussed in the next section.

Figure 6.4 shows an example of an "is-a" relation between two Reference Frames. On the left side, the figure shows the Palette of a Reference Frame in the domain of stochastics, the right side of the figure contains the user interface of an extended stochastics Reference Frame. It is visible that the right Reference Frame extends the left one with respect to both node and edge types - syntax and semantics are also retained in this extension, although this has of course no visual representation in the figure.

Besides illustrating the concept of "is-a" relations between Reference Frames, this example also shows some advantages of the mechanism. The reduced complexity in "basic" versions of Reference Frames may help users to understand the

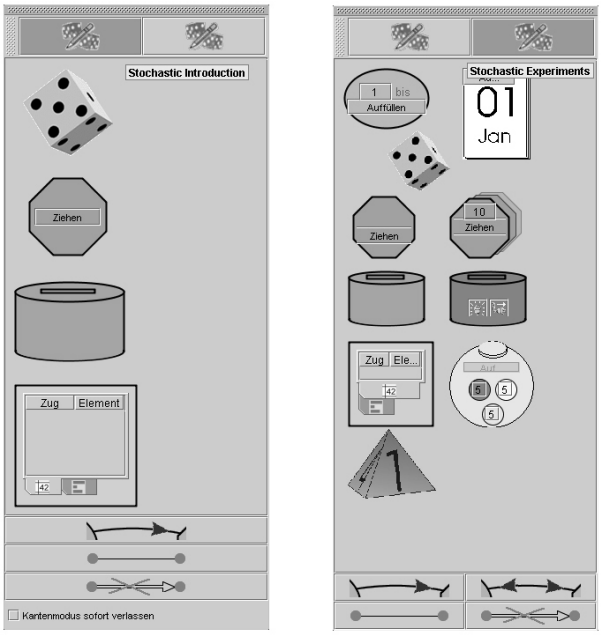

Figure 6.4: The user interface of a Basic and an Extended Reference Frame for stochastics

principles of the modeling language, while advanced users can switch to more expressive and powerful versions. Apart from these usability issues, there are also some education-related advantages of iterative versions: they allow the teacher to adjust the level of modeling language abstraction according to his pupils and their learning advances. This is visible in figure 6.4: the "basic" version of the Reference Frame contains very simple elements to model random processes: an urn, a dice, a representation of "drawing", and a result visualizer. The extended version offers a whole range of more advanced concepts, including abstract urns, elements for the repetition of drawings, and edges that model the drawing elements with putting them back immediately.

6.4 Event Based Model Interpretation

The previous parts of this chapter described the computational representation of visual typed graphs and Reference Frames. Building upon some approaches presented by Pinkwart, Hoppe, Bollen, and Fuhlrott (2002), this section describes how the interpretation of visual typed graphs by a set of Reference Frames can practically be implemented.

Here, one of the foundational design decisions is the interaction model between the visual typed graph and the Reference Frames. Two classical alternatives are the *pull model* (in which the Reference Frames would query the visual typed graph, usually interval-based), and the *push model*. In the latter, the visual typed graph is an active structure which propagates changes to the Reference Frames.

The push model has a number of advantages: it reduces the total number of messages (as no messages are sent as long as the data stays unchanged), and is faster in providing the information receiver with data (as time intervals do not play a role). The COLLIDE JGraph supports the push model (cf. subsection 5.2.4), so that its adoption as a base for both the syntax checking and the model

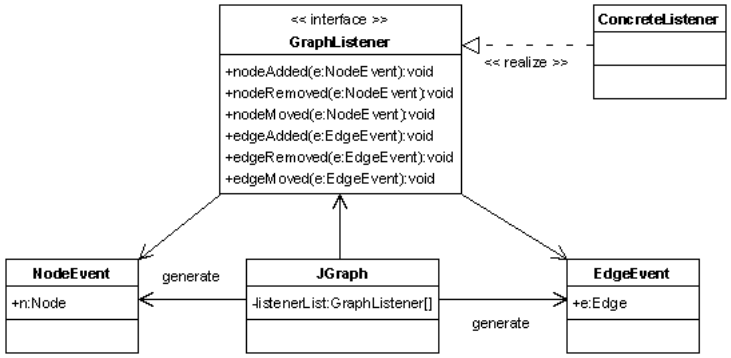

Figure 6.5: The event classes in the COLLIDE JGraph architecture

interpretation makes sense. The basic event types employed for both tasks are `NodeEvent` and `EdgeEvent`. As shown in figure 6.5, these events can be handled by implementations of the `GraphListener` interface, which consists of methods that represent structural changes in visual typed graphs. Typical listeners are the `ReferenceFrame` implementations or delegates, and also nodes and edges. The next subsections show these listeners (`ConcreteListener` class in the diagram) can be designed and connected in order to manage syntax and semantics of models.

In the runtime system, `NodeEvent`s and `EdgeEvent`s events are produced whenever the abstract graph structure has changed, which may have one of the following causes:

- the user manipulating the visual typed graph,
- remote synchronized applications whose visual graph changes, or
- other local system-internal processes initiated by Reference Frames (in particular, the interpretation of a visual typed graph is allowed to change the interpreted graph!)

6.4.1 Syntax

As argued in section 4.6, the only reasonable way of defining a visual typed graph G syntactically correct with respect to a set of Reference Frames R is to base this on the syntax predicates defined by the Reference Frames contained in R. Consequently, G is then correct concerning R if and only if G is correct in terms of each single $\mathcal{R} \in R$.

Therefore, an implementation which ensures syntactic correctness of visual typed graphs by sets of Reference Frames has to guarantee that at any point in time, all integrity constraints hold, i.e. all mappings evaluate to `true`. This is possible with a simple algorithm outlined in the state diagram in figure 6.6.

Proposition 6.2 *Assuming a correct constraint checking mechanism for visual typed graphs, the algorithm from figure 6.6 guarantees syntactic correctness of a visual typed graph with respect to a set of Reference Frames.*

Proof. If the set R of Reference Frames is empty, then there are no constraint mappings, which leads to trivial fulfilment of the syntactic correctness criterion. Whenever an element is added to R, the constraint mappings defined by this Reference Frame are transmitted to the graph G (state "Calculate constraints", event

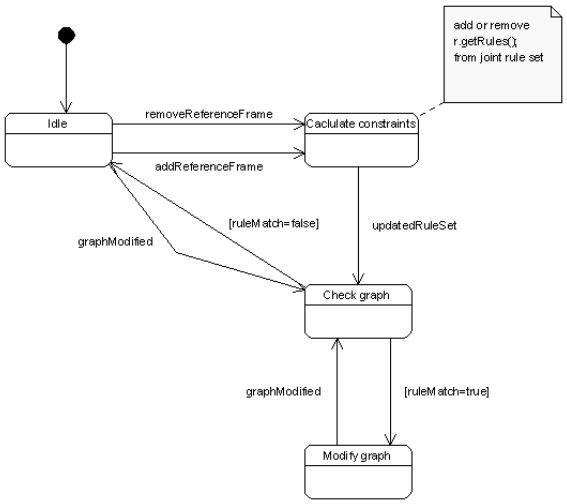

Figure 6.6: Basic algorithm for ensuring syntactic correctness of visual typed graphs with respect to multiple Reference Frames

"updatedRuleSet"), and therefore (by assumption) hold, at least after potential modifications within G (state "Modify graph"). If an element is removed from R, syntax problems cannot occur, as the total set of constraint mappings is reduced. Finally, with R temporarily stable, the result of any modification within G is (by assumption correctly) checked with respect to the constraint mappings as defined by the elements of R, so that only change attempts of G that yield a syntactically correct state are accepted.

This algorithm is implemented in the Cool Modes framework (cf. next section). Here, a `FrameController` holds a list of currently available Reference Frame classes, and a `JGraph` represents a visual typed graph, including also the ability to check constraints in form of rules (cf. sections 6.1 and 6.2). The algorithms for checking the matching of rules with the graph have to face the problem that graph isomorphism is a computationally hard problem (Skiena, 1990). This theoretical problem, however, is not a practical one for three reasons: First, the number of nodes and edges (i.e., the *complexity* of the graph) is usually not too high in the targeted modeling situations (as the users need to have an understanding of the model). Second, the *type* information associated to nodes, edges, and rules greatly reduces the amount of potential matches for rules to be checked. Third, the events that are sources for constraint checks include the source of the event. This can be used to further restrict the scope of the matching routines.

The described algorithm to ensure the syntactic correctness of a visual typed graph with respect to multiple Reference Frames can be modified in some directions, taking into account inheritance relations between `ReferenceFrame` implementations. A *history-preserving* variant of the algorithm is shown in figure 6.7. Here, upon adding a Reference Frame class to R, also the superclass is added to R, provided that it is also a Reference Frame (state "Check superclass").

This history-preserving variant of the algorithm has one central advantage:

Proposition 6.3 *The history-preserving algorithm for syntax preservation ensures that all class-level Reference Frame extensions are treated like "is-a" extensions (in the sense of definition 4.13) by the constraint check.*

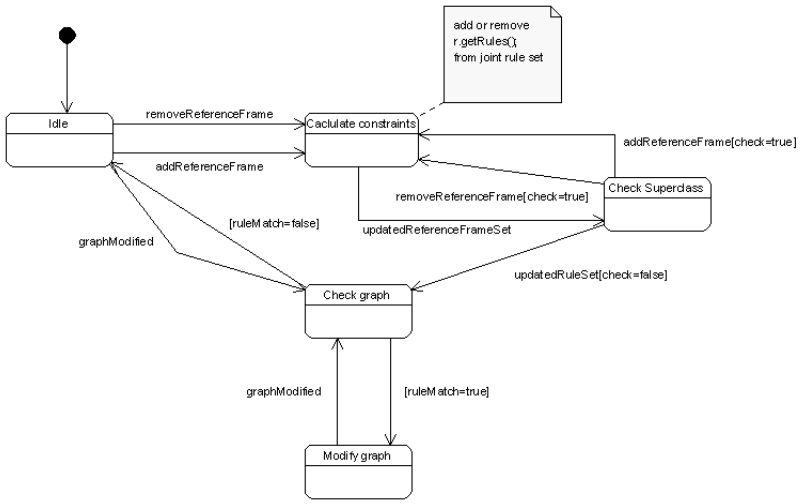

Figure 6.7: History-preserving variant of the algorithm for syntactic correctness

Proof. One result of the discussion in section 6.3 was that a subclass of a Reference Frame class is in an "is-a" relation to its superclass as long as the constraint mappings set defined by the subclass includes the one defined by the superclass. In the history-preserving algorithm variant, this property is guaranteed by explicitly adding the constraint mappings of the superclass to those that are applied for checking syntactical correctness.

Of course, proposition 6.3 does not address code-level issues between the classes: here, the constraint mapping sets may still not fulfil the criteria for "is-a" relations. However, the algorithm ensures (or: emulates) the *effects* of "is-a" relations at runtime. This is similar for the strictly history-preserving variant of the syntax preservation algorithm, which is shown in figure 6.8. This includes the following add-on in the state "Filter rules": if a class `c` and its superclass `c'` are both in R, then before proceeding to the "Check graph" state (i.e., applying the rules), those constraint mappings defined by `c` are removed from the set that only use node and edge types defined by the `getNodeAssociations()` and `getEdgeAssociations()` methods of `c'`.

Proposition 6.4 *The strictly history-preserving algorithm for syntax preservation ensures that all class-level Reference Frame extensions are treated like syntactically consistent extensions (cf. definition 4.14) by the constraint check.*

Proof. Assume that G is a visual typed graph which only consists of node and edge types defined in a `ReferenceFrame` class `c'`, and let `c` be a subclass of `c'`. Furthermore, assume that G fulfils all the constraint mappings defined by `c'`.

Adding `c` to R leads to adding also `c'` to R, due to the history-preservation. As all constraint mappings of `c` that operate only on node types defined by `c'` are filtered out, all the constraint mappings transmitted from `c` to G evaluate to `true`, so that G is (by assumption) evaluated as syntactically correct.

The two previous propositions outlined how, with minor changes in the algorithm, remarkable effects can be reached. However, it is difficult to judge about a "best" variant of the algorithm: on the one hand, the proposed variants ensure certain coherence properties that the pure Reference Frame implementation cannot

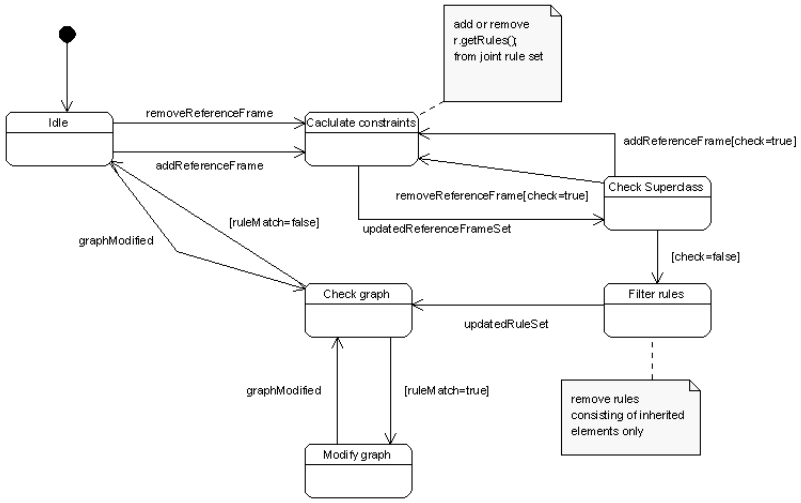

Figure 6.8: Strictly history-preserving variant of the algorithm for syntactic correctness

guarantee. On the other hand, they reduce the design space for a Reference Frame: programmers might want to use the inheritance mechanism known to them without the side-effects that the algorithm variants introduce. Also on a conceptual level, inconsistent extensions might indeed be *wanted*. E.g., the definition of a simple Reference Frame that defines only some node and edge types but introduces neither syntax nor semantics allows users to build arbitrarily, uninterpreted structures with the primitive entities provided by that Reference Frame. An extended Reference Frame, which adds only syntactic constraints and interpretation mappings, can then allow the users to make use of further functions, e.g. for *simulating* the structures in some sense. However, such an extension is not consistent and would be prevented by the strictly history-preserving algorithm. For that reason, the Cool Modes system can be parameterized so that different variants of the syntax preservation algorithm are possible.

6.4.2 Semantics

The previous subsection described how the syntax constraint mappings defined by Reference Frames can be checked for a visual typed graph under the condition of a dynamic set of Reference Frames and a modifiable visual typed graph. As shown, the syntax algorithm is centralized in the way it manages the set of constraint mappings.

With syntactical correctness guaranteed, the next important topic is how the interpretation of visual typed graphs by multiple Reference Frames (cf. subsection 6.3.1) can be implemented.

Interpretation by single Reference Frames

As expressed in subsection 6.3.1, the classes that represent Reference Frames and the classes that represent node or edge types known by these Reference Frames can have attributes whose domain corresponds to the semantic domain, and operations that calculate values for the attributes (and thus correspond to the semantic mapping). However, subsection 6.3.1 did not go into detail about the mechanisms that enable the *invocation* of the semantic mapping calculation.

```
public interface NodeListener {
    public void edgeAdded(NeighborEvent e);
    public void edgeRemoved(NeighborEvent e);
    public void stateChanged(NeighborEvent e);
}
```

Figure 6.9: The technical NodeListener interface

Similar to the case of syntax checking, the semantic mapping is based on a push model with actions and events. This is reasonable, since the value of a semantic attribute will only have to be re-calculated upon a change in the visual typed graph - provided that the notion of "change" is kept broad enough to cover all possible event sources (cf. list of action sources at the beginning of this section).

Compared to the proposed model for verifying the syntax of visual typed graphs (which relies on a centralized management), the information transfer model employed for the semantic mapping needs to be more flexible for several reasons:

1. The options for a Reference Frame to interpret a visual typed graph need to be very flexible, in order not to restrict the set of modeling languages that can be expressed in a Reference Frame. Therefore, a central "interpreter" instance (which is functionally richer than a typical Broker component) does not make much sense.

2. The events used for syntax checking are not sufficient: additional events need to be generated whenever the semantics of an element (i.e., the value of a semantics-related attribute) changes, as this can cause further changes

3. With more event types and more receivers than needed for the case of syntax checking, the total number of messages would significantly increase without further reduction or filter techniques.

4. Not all Reference Frames need to be informed about all events: e.g., if a certain node type is not in the KNOWS set of a Reference Frame, then a change of *semantics* related to a node of this type will usually *not* have to be transmitted to this Reference Frame - yet, *structural* changes in the graph (e.g., adding nodes) might typically be triggers to start the generic interpretation mechanisms. However, due to the degrees of flexibility in defining Reference Frame classes (and, in particular, the implicit way of declaring semantics attributes and imported node and edge types), there is no way for a an external component to automatically calculate what events have to be distributed to which sets of Reference Frames.

A solution to these problems is to base the event propagation mechanism concerning semantics on the Publisher Subscriber model (Buschmann et al., 1996). In this approach, events are only sent to a component after a "subscription" (which solves problem 3), the interested component can subscribe itself as listener (problem 4), and the interpreter can be of any type, provided that the class implements the corresponding interface (which eliminates the need of a central interpreter, problem 1). A new event type (not used within the syntax check mechanism) can be used to address problem 2. As shown in figure 6.9, a `NodeListener` implementation can be informed about state changes within a node (an analogous class exists for edges). The JGRAPH library has been extended with a number of help routines that facilitate the subscription (either to all events in the graph, or to events related only to selected nodes or edges).

Using these interfaces, a Reference Frame can react upon changes in the state of nodes and edges (in particular with respect to attributes that relate to the semantics), and re-calculate the semantics of the whole visual typed graph.

It should be noted that there is no policy about *which* components should act as listeners for state changes. This leaves design space and enables both "local" and "global" methods for determining model semantics:

Local Methods. These methods do not require a central "model interpreter", not even on the level of the Reference Frame. Typically, the `ReferenceFrame` implementation class does therefore not contain any semantics-related methods or attributes - however, the node and edge classes do. Two examples that underline the usefulness of local methods are the interpretation of Petri Nets (here, the activation states of transitions and the effects of firing them can be calculated locally within the transition classes, which only need to know their connected places), and the code generation from UML class diagrams. Also in the UML case, the code of a class or interface that is represented in the diagram can be generated solely based on information about this element and its neighbors in the graph.

Global Methods. In contrast to the local methods which are decentralized and contain the semantic mapping in the node and edge classes, *global methods* make use of one central interpretation component - which can typically be located in the `ReferenceFrame` implementation class in my approach. Examples of modeling languages which are well suited for global approaches include those that make use of spatial relations (like, e.g., puzzles), and those that need to calculate new values for a whole number of sematic attributes simultaneously, instead of adopting a "data propagation through the graph" approach. Here, an example is the System Dynamics modeling method.

The following proposition is an immediate consequence of the chosen design. Its proof is nearly trivial - yet, the proposition reflects the distinctive factor between the proposed implementation of the Reference Frame approach, and most of the other modeling systems reviewed in chapter 3: with the exception of some (non-collaborative) metamodeling systems, the tools usually restrict the expressiveness of the supported modeling languages in some way if dynamic adding of languages is supported at all. This is not the case for the implementation proposed in this thesis:

Proposition 6.5 *The model interpretation possible in the Reference Frame implementations is as expressive as the Java programming language.*

Proof. A Reference Frame implementation can attach itself as listener to any event in the model graph. Thus, it gets notified whenever any modification within the model occurs. A notification consists of a method invocation, and is thus an entry point into a java program. As a Reference Frame is not contained in any sandbox but has full application rights, the proposition is proved.

A constructive proof is also possible by defining a Reference Frame which provides a node type suitable for entering Java program code, and offers the possibility to compile and execute this code. Such a Reference Frame has been developed by Baloian, Pino, and Motelet (2003), who have used the implementation within a Java course.

Interpretation by multiple Reference Frames

In the described approach, each Reference Frame is autonomous in interpreting visual typed graphs, and there are a variety of options for implementing the interpretations based on events of the type `GraphEvent`, `NodeEvent`, and `EdgeEvent`.

In the Publisher Subscriber pattern, which is adopted for the event propagation, more than one Reference Frame interpreting a visual typed graph simply translates to multiple listeners on an event source and is therefore generically supported. Thus, in terms of subsection 4.6.3, the process of managing multiple interpretations is invoked upon any event (of the described types) that occurs within the visual typed graph. In the proposed implementation, the $\otimes$ operator is of the "integration" type, and results in an implicity defined μ, which is an aggregation of interpretations. In terms of section 4.6.3, this can be expressed with the following conceptual relation:

$$Ip(\langle G, \mathcal{N}, \mathcal{E}, L\rangle) = Ip(\mathcal{R}_1)(\langle G, \mathcal{N}, \mathcal{E}, L\rangle) \circ \ldots \circ Ip(\mathcal{R}_n)(\langle G, \mathcal{N}, \mathcal{E}, L\rangle)$$

Here, $\mathcal{R}_1$ to $\mathcal{R}_n$ correspond to the Reference Frames that *are registered as listeners* on the element in $\langle G, \mathcal{N}, \mathcal{E}, L\rangle$ that conceptually causes the event upon which the interpretation is invoked. Thus, the μ operator is not only *implicitly* defined, but also *partially*: the interpretations to be aggregated vary with the listeners defined by the Reference Frames. It is even possible that a Reference Frame does not attach any components as listeners, and therefore does not conduct any interpretation. Furthermore, the order of interpretations is not fixed and depend on the order of the registration as event listener (cf. discussion in the following of this subsection).

An interesting side aspect that comes with the possibility of storing interpretation results (i.e., semantics) either in the Reference Frame or in node/edge classes is that the former is not accessible to other Reference Frames, in contrast to the latter. In terms of subsection 4.6.3, this leads to the option of ensuring the *separation* of interpretation results by storing these in the `ReferenceFrame` class: in this case, the aggregation operator $\circ$ does, for this particular interpretation, only have a symbolic and formal meaning. On the other hand, all the interpretation results stored in the node and edge classes are subject of potential modification by other interpreters.

Similar to the aggregation operator μ, also the difference between generic and domain specific interpretation is implicitly defined in the proposed approach: as stated, a Reference Frame can subscribe as listener for any changes in the visual typed graph, including also the parts of the graph that are "unknown". Here, the access to semantics-related attributes of nodes and edges is obviously a specific case of the Reference Frame importing the corresponding node/edge type (cf. subsection 6.3.1 and the implicit definition of imports). Therefore, a trivial deduction is that in the proposed implementation, the domain specific interpretation done by a Reference Frame is limited to its KNOWS set, whereas the generic interpretation is enabled for the whole visual typed graph (cf. subsection 4.6.3). Here again, the $\otimes$ in the formal $Ip_{gen}(\mathcal{R})(\langle G, \_, \_, L\rangle) \otimes Ip_{dom}(\mathcal{R})(\langle G_{|\mathcal{R}}, \mathcal{N}, \mathcal{E}, L\rangle)$ notation translates to an aggregation operator. On the implementation level, the event type (structural change vs. semantics change) distinguishes between generic (only structural change events) and domain specific (all events) interpretations - an integrated treatment of these events is possible though not enforced. The following scheme outlines the event distribution to multiple Reference Frames.

1. Any Reference Frames R can subscribe as an event listener for elements (nodes, edges, or the whole graph) of a visual typed graph G. This subscription be done or removed at any point in time.

2. Upon a change in its graph structure that leads to a syntactically correct state (cf. figure 6.6), G sends out change events to the Reference Frames that are registered as listeners on G.

3. Nodes and edges in G can submit changes in their internal states to G.

4. Upon the reception of a state change event in a node n, or a change in the neighbor areas of n in G, G sends out change events to the Reference Frames that are registered as listeners on n.

5. Upon the reception of a state change event in an edge e, G sends out change events to the Reference Frames that are registered as listeners on e.

6. The reception of a change event invokes the model interpretation mechanism in the Reference Frame.

One not explicitly defined aspect in this algorithm is the method used by the graph for publishing changes. Here, two different approaches are possible - both have some disadvantages:

Synchronized. Invoking the listeners using synchronous method calls requires (or: induces) an *order* among the listeners. As the execution of interpretation routines (the $\circ$ operator) is generally not commutative, this order may have an impact on the outcome of the interpretation. Furthermore, a "best" order is not easy to define in the case of heterogeneous models (cf. subsection 4.6.3). Apart from this order problem, the synchronized method makes the overall interpretation mechanism critically dependent on the algorithms that the single Reference Frames provide: if one of has, e.g., an infinite loop, the whole mechanism hangs.

Concurrent. The use of asynchronous techniques for invoking the listeners does not have the blocking risk. Yet, it is even worse in terms of interpretation results, as (without further concurrency control mechanisms) it allows Reference Frames to interpret parts of models without granting them that the model stays stable during the runtime of the interpretation algorithm. This might easily lead to an interleaving of interpretations which is not serially equivalent to *any* of the orders used within a synchronized approach.

As the modification of the visual typed graph as a result of the interpretation process is generally allowed (and necessary for modeling purposes), both approaches have the risk of infinite event loops, even in the case of only one Reference Frame. These loops are easy to detect, since there is a single point of event distribution: despite this, the decision whether to prevent event loops or not is not easy, as Reference Frames (or sets of interacting Reference Frames) might use these event loops as a means of simulating a model (and provide internal termination rules).

Currently, the JGraph implementation does not disallow event loops and supports only the synchronized approach of message distribution (with the order of notifications according to the order in which the subscriptions were made). Yet, the concurrent approach could easily be integrated, as well as event loop checks.

6.5 Interoperability Issues and Design Aims Met

This chapter of the thesis presented one possible implementation of the Reference Frame approach as described in chapter 4. In accordance with some implementation recommendations related to the meta modeling techniques shown in section 2.3, the

presented approach adopts an object oriented approach for representing both models and modeling languages.

An explicit aim of this chapter was to "put into practice" the Reference Frame approach, and describe visual typed graphs and Reference Frames not only on a conceptual, but also on an operational level. Therefore, the chapter contains remarks about general architectural aspects, descriptions how to make use of the existing COLLIDE JGraph library, outlines of specific algorithms, details about needed events and central interfaces, and also discussions of some design alternatives and their consequences.

As discussed in section 1.5, the proposed implementation framework puts an explicit focus on three things: *simplicity* in use (here, for the programmer who intends to develop Reference Frames), *flexibility* in the sense that the scope of supported modeling languages is not unnecessarily restricted, and *interoperability*. The design aim of simplicity has lead to some elements of the conceptual framework being defined only implicitly. Examples are the visual attributes and the algorithm for multiple integrated interpretations. Some other elements, like the synchronization contexts, still have to be explicitly defined by a Reference Frame implementation. To assist users in developing these, the proposed implementation includes a number of service classes and libraries which cover a range of "typically needed" functions. In accordance with the design claims of Roschelle et al. (2000), programming conventions are made explicit, and design patterns are used to help the user understand the interrelations within the system, and the functionality of service classes and interfaces.

This chapter discussed a number of critical design choices, which can in some cases be regarded as tradeoffs between flexibility and interoperability on the one hand, and security or guaranteed functionality on the other hand. Here, one example is the order of event receivers in the event distribution scheme employed for integrated interpretations, and in particular the check for loops in this scheme. Obviously, the proposed "loose control" approach (in contrast to, e.g., assigning fixed orders by ID's and generally preventing loops) has weaknesses in the sense that it may theoretically lead to undesired or incoherent integrated interpretations. However, this has to be seen within the general discussion of integrating arbitrary modeling techniques on a semantic level, for which an easy-to-use *and* generally applicable technique currently seems out of reach (cf. chapter 2 and in particular section 2.3.4). In contrast to both theoretical approaches like, e.g., structured modeling (Geoffrion, 1989a) and also a number of comparable systems like, e.g., Ptolemy or ModellingSpace who solve these "critical" parts by reducing expressiveness and/or interoperability, the implementation proposed in this chapter relies to some extent on the developer of Reference Frames. He has a high degree of flexibility in importing (and thus re-using) elements from other modeling languages, or even whole languages. The system allows for heterogeneous models and multiple interpretations - yet, as the syntax and semantics that a developer assigns to "imported" structures is (on purpose!) *not* generally checked for total conformance with the "original", conflicts might occur. It is the task of the Reference Frame developer to consider them - under the assumption that re-use of elements or functionality is done on purpose, this delegation of responsibility is not unreasonable. As discussed in this chapter, alternatives to this are possible, but would typically restrict flexibility and/or interoperability of the approach.

Within the discussion of challenges and aims for this thesis in section 1.5, the criteria list of Dolk and Kottemann (1993) (page 19) played an important role. With the exceptions of the user interface (which is not in the scope of this chapter), and the model solution library (which is not an intended target of this thesis), the Reference Frame framework as presented in this chapter fulfils their criteria:

- With the `ReferenceFrame` interface and the `JGraph` class, there are uniform internal schemes that are capable of representing many classes of models and modeling languages.

- Conversion of external representations is easily possible by translators and wrappers, as `ReferenceFrame` implementations have the full expressiveness of the Java programming language.

- The object oriented approach in Java allows for robust typing and inheritance options at various levels.

- The action/event based approach is a suitable foundation for supporting dynamic modeling with active and interactive structures (Lenard, 1993).

Some of these points also relate to the criteria listed by Roschelle et al. (2000) (page 20). In particular, the "dynamic publishing and subscribing mechanisms" and the "change coordination patterns" that they demand are available with the chosen way of information distribution (cf. section 6.4). The advanced component persistence mechanisms they claim are already basically supported by the COLLIDE JGRAPH library and its XML storage format (the next chapter will discuss further persistence issues). Also a number of points listed by Roschelle et al. (1999) (page 20) are fulfilled: options for component re-use are available (e.g. with importing elements into Reference Frames), translators and wrappers embedded in the code of Reference Frames (or even nodes or edges) can adapt external resource formats to internal ones, and the flexible event-based model interpretation algorithm provides a whole range of "wiring" options between elements, which facilitates interoperability.

Going into the details of interoperability support in the proposed implementation, the dimension of *syntax* is addressed through three aspects: First, an important point is that the implementation does fully support heterogeneous visual typed graphs in the sense of chapter 4. Second, the placement of visual attributes in interfaces and abstract base classes (cf. subsection 6.1.2) ensures their independency of Reference Frames. This is conform with the approach taken throughout chapter 4. The need for a minimum set of visual attributes required for displaying the visual typed graphs is taken into account by including corresponding methods in the central interfaces for nodes and edges. Finally, the algorithms for checking syntactic correctness as discussed in subsection 6.4.1 are a central contribution to syntactic interoperability.

On the level of *semantics*, interoperability has been addressed in this chapter primarily by two means. The first deals with the relations between Reference Frames. Here, the framework provides a whole range of options for re-use of elements and functionality, and in particular also for sharing of semantics and interpretations in the integrated environment. The two techniques (import and is-a) presented in subsection 6.3.2 exemplify possible interrelations between Reference Frames and connect to the conceptual relations developed in subsections 4.5.1 and 4.5.2 - however, on the code level, also other types of re-use are of course possible. The second dimension of semantic interoperability is related to the technique for integrated interpretation of heterogeneous models, described in subsection 6.4.2. Based on the Publisher Subscriber design pattern, the algorithm offers one possible implementation of the μ operator introduced in subsection 4.6.3. The presented approach enables the storage of model semantics in nodes and edges objects or in the Reference Frame. These two options allow for a separation of individual interpretations as well as for real integrated and "shared" solutions.

On the abstract framework level that this chapter addresses, one aspect of *task interoperability* is covered: as argued, the implementation allows for using different

Reference Frames in an integrated manner, and for constructing heterogeneous visual typed graphs. As Reference Frames can be conceived as modeling languages or techniques, this integration feature can avoid tool breaks, and thus provide support for tasks that consist of different phases (each of these associated to certain tools).

Chapter 7

The Cool Modes Framework

In the three previous chapters of this thesis, Reference Frames as a conceptual base for collaborative modeling systems have been presented, and an abstract implementation based on existing software libraries has been proposed. This chapter concludes the implementation parts of this thesis with the presentation of the collaborative modeling tool Cool Modes (COllaborative Open Learning and MODEling System) as an example application which puts into practice the Reference Frame ideas and architectures outlined before, considering the interoperability issues raised in section 1.5.

In particular, this chapter discusses *specification strategies* and *user interfaces* for Reference Frames, as well as issues dealing with *collaboration* support in the context of modeling with Cool Modes.

7.1 Definition and Usage of Reference Frames

This section discusses three different approaches of specifying/developing Reference Frames, and shows how the runtime management of these Reference Frame implementations is done in Cool Modes.

7.1.1 Definitions of Reference Frames

Simplicity in system usage is always a criterion worth considering. For the case of the collaborative modeling framework as targeted within this thesis, this simplicity can be seen from two different perspectives: the *use* of the modeling environment as such, and the *development* of Reference Frames (modeling languages) together with their integration into the environment. For the case of educational applications, this aspect has in particular been demanded by Roschelle et al. (1999), who claim that the integration of new components into interoperable frameworks should be as easy as copy&paste operations.

Using the interfaces and abstract base classes for Reference Frames as described in the previous chapter, there are several options of implementing concrete Reference Frame implementations. Three of these, as integrated in the Cool Modes framework, are shortly summarized and compared in the following.

Program centered Approach

As shown in section 6.3, the proposed architecture represents a Reference Frame via the interface `info.collide.frames.ReferenceFrame`. The abstract Java adapter class `AbstractReferenceFrame` offers default implementations for most of the interface methods.

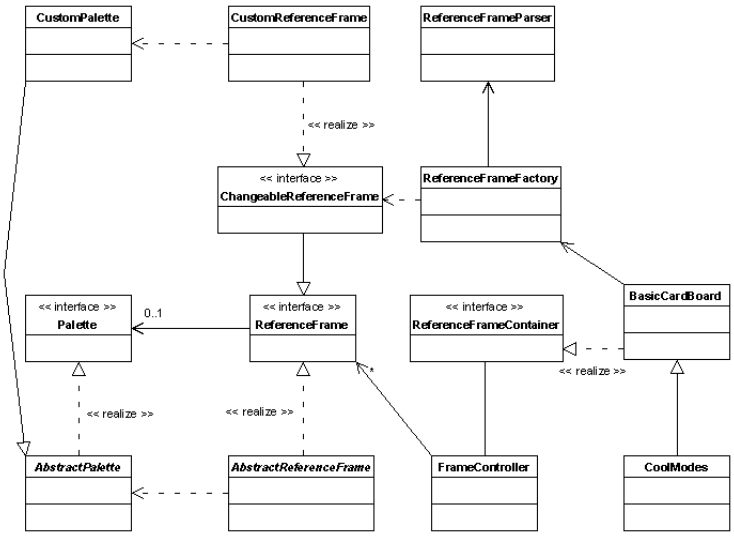

Figure 7.1: Reference Frames and Palettes in the Cool Modes system architecture

From an object oriented programming perspective, a natural way of defining a concrete Reference Frame is the development of a class which implements the mentioned interface, either directly or indirectly through inheritance from the abstract adapter class. This program based way is fully supported in the Cool Modes environment. Subsection 7.1.2 shows how these Java based Reference Frame definitions are retrieved and managed by the runtime system.

Document centered Approach

As an alternative to the "straightforward" program based method of Reference Frame definitions outlined before, the Cool Modes environment offers the option of describing Reference Frames in a document based way using XML files. To enable the integration of these data files with the architecture presented in section 6.3, the framework contains template classes which are parameterized with the XML data. Figure 7.1 illustrates these classes in their architecture context: the interface `info.collide.frames.ChangeableReferenceFrame` identifies template Reference Frames (which differ from "normal" ones in that they have additional methods to allow the setting of values), the class `info.collide.frames.CustomReferenceFrame` is a concrete implementation of this interface. Finally, parsers and Factory classes complete this part of the system design.

Figure 7.2 shows the document type definition for XML specifications of Reference Frames (leaving out some lower level entities that are connections to the JGraph XML format). Most of the data elements contained in the DTD have direct corresponding methods in the Java interface for Reference Frames shown in figure 6.1. In particular, the document centered specification relies on the following principles:

- An XML representation of a Reference Frame contains the specification of a Java template class to be parameterized with the XML file.
- Node and edge type definitions can either be done by
  - explicitly referring to the class name of the controller class and the model

```
<!ELEMENT ReferenceFrame
         (TemplateClass?,Objects,Palette?,Metadata)>

<!ELEMENT TemplateClass (#PCDATA)>

<!ELEMENT Objects (Node*,Edge*,Rule*,SyncContext?)>
<!ELEMENT Node ((ClassName,ClassName)|NodeModel|NodeRef)>
<!ELEMENT Edge ((ClassName,ClassName)|EdgeModel|EdgeRef)>
<!ELEMENT ClassName (#PCDATA)>
<!ELEMENT NodeModel ANY>
<!ATTLIST NodeModel controllerClass CDATA #REQUIRED>
<!ELEMENT EdgeModel ANY>
<!ATTLIST EdgeModel controllerClass CDATA #REQUIRED>
<!ELEMENT NodeRef EMPTY>
<!ATTLIST NodeRef className CDATA #REQUIRED>
<!ELEMENT EdgeRef EMPTY>
<!ATTLIST EdgeRef className CDATA #REQUIRED>
<!ELEMENT Rule (EgdeRule|CycleRule|PatternRule)>
<!ATTLIST Rule Message CDATA #IMPLIED>
<!ELEMENT EdgeRule (NodeRef,NodeRef,EdgeRef)>
<!ATTLIST EdgeRule weight CDATA #IMPLIED
                   limitFrom CDATA #IMPLIED
                   limitTo CDATA #IMPLIED
                   limitMessage CDATA #IMPLIED>
<!ELEMENT CycleRule (EdgeRef*,NodeRef*)>
<!ELEMENT PatternRule (JGraph)>
<!ATTLIST PatternRule type (structure|equality) "structure">
<!ELEMENT SyncContext EMPTY>
<!ATTLIST SyncContext
          type (single|connectivity|induced|graph) "single">

<!ELEMENT Palette
          (PaletteClassRef|(PaletteTemplateClass?,
          Icon?,ToolTip?))>
<!ELEMENT PaletteClassRef (#PCDATA)>
<!ELEMENT PaletteTemplateClass (#PCDATA)>
<!ELEMENT Icon (#PCDATA)>
<!ELEMENT ToolTip (#PCDATA)>




<!ELEMENT Metadata
         (Package*,Name*,Author*,Language*,NeededResource*)>
<!ELEMENT Package (#PCDATA)>
<!ELEMENT Name (#PCDATA)>
<!ATTLIST Name language CDATA #IMPLIED>
<!ELEMENT Author (#PCDATA)>
<!ELEMENT Language (#PCDATA)>
<!ELEMENT NeededResource (#PCDATA)>
<!ATTLIST NeededResource internal (true|false) "true">
```

Figure 7.2: Document type definition for data based Reference Frame descriptions

class (to generate the required `Association` objects at runtime, cf. subsection 6.3.1), or by

– specifying the class of the controller and a `NodeModel` or `EdgeModel` element, which represents the model. This approach uses the XML serialization functions of the nodes and edges in the COLLIDE JGraph library in order to parameterize the node and edge classes available in Reference Frames.

- Node and edge types can be imported using the NodeRef and EdgeRef elements.
- The synchronization context mapping can be chosen from the four strategies discussed on subsection 6.3.1.
- Using, again, a template class mechanism, Palettes (user interfaces for Reference Frames, cf. section 7.2) can be specified by referring to classes that generate a "default" user interface which contains the elements defined by the Reference Frame - in particular, the node and edge types.
- An implicit condition is that template classes specified for Reference Frames and Palettes have to fit in the framework in the sense that they implement the respective interfaces. Otherwise, the runtime environment uses default template classes.

Figure 7.3 shows a simple example of a document centered definition of a Reference Frame for Petri Nets. The definition makes use of the first way of node and edge specification (i.e., passing model and controller classes), and sets the default of the synchronization contexts to "connectivity component" (this is not the minimal solution, but an appropriate choice based on the four available templates). Two rules are outlined (ensuring the bipartiteness property), and template classes for the Reference Frame and its Palette are chosen. For reasons of space, the figure does not contain the metadata of the Reference Frame, and also the package information in the class references is omitted.

Mixed Approaches

Apart from the "pure" program centered and document centered approaches for Reference Frame specification, intermediary solutions are possible. Within the description of the XML based approach, the option of referring to Java classes serving as templates to be parameterized with the XML file has already been outlined. This option is available both for the Reference Frame and its Palette, which enables one form of a "mixed" Reference Frame specification style, consisting of a main XML file and a set of associated Java classes.

Other mixed specification forms are possible with the program based approach. Here, one option is the inclusion of a `JGraph`, serving as a source for drag&drop operations, in the Palette. The graph itself can be loaded from an XML file, using the serialization functions of the COLLIDE JGraph library. Also in this approach, a mixture of Java and XML files constitute a Reference Frame - yet, the "main" element is a class file here.

Comparison

The program and document centered approaches for defining Reference Frames have one point in common: the developer has to adhere to a certain structure in order to allow for an integration of the Reference Frame into the environment. This interface

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ReferenceFrame SYSTEM "Frame.dtd">
<ReferenceFrame>
 <TemplateClass>CustomReferenceFrame</TemplateClass>
 <Objects>
  <Node>
   <ClassName>PlaceNode</ClassName>
   <ClassName>PlaceNodeModel</ClassName>
  </Node>
  <Node>
   <ClassName>TransitionNode</ClassName>
   <ClassName>TransitionNodeModel</ClassName>
  </Node>
  <Edge>
   <ClassName>PetriEdge</ClassName>
   <ClassName>PetriEdgeModel</ClassName>
  </Edge>
  <Rule Message="Do not connect places to places!">
   <EdgeRule weight="0">
    <NodeRef className="PlaceNodeModel"/>
    <NodeRef className="PlaceNodeModel"/>
    <EdgeRef className="PetriEdgeModel"/>
   </EdgeRule>
  </Rule>
  <Rule Message="Do not connect transitions to transitions!">
   <EdgeRule weight="0">
    <NodeRef className="TransitionNodeModel"/>
    <NodeRef className="TransitionNodeModel"/>
    <EdgeRef className="PetriEdgeModel"/>
   </EdgeRule>
  </Rule>
  <SyncContext type="connectivity"/>
 </Objects>
 <Palette>
  <PaletteTemplateClass>CustomPalette</PaletteTemplateClass>
  <Icon>petri32.gif</Icon>
  <ToolTip>Petri Nets</ToolTip>
 </Palette>
 <Metadata> ... </Metadata>
</ReferenceFrame>
```

Figure 7.3: Example of an XML based Reference Frame definition

is a Java interface for the program centered approach, and a DTD for the document centered approach.

The specification in XML is easier (and can be the hook for a "Reference Frame editor", cf. chapter 9), and does not require Java programming skills as long as the existing template classes for Reference Frames and Palettes are sufficient, and suitable node and edge classes for parametrization are available.

The limits of the XML based approach, however, lie in its inherent representation of Reference Frames as passive *data*: in contrast to the program centered approach, a flexible specification of *runtime behavior* (e.g., algorithms to simulate models) is not possible. Due to this, the options for document centered specification of Reference Frames are severely limited in their expressiveness concerning semantics. The other elements of the formal framework developed in section 4.5 are supported - yet, only by means of choices between *existing* objects (e.g., node and edge types) or algorithms (in the case of the synchronization context mapping).

In terms of the Model View Controller scheme as used within the COLLIDE JGraph library, the limits of the document centered specification have been explored by Pinkwart, Hoppe, and Gaßner (2001). Here, one result is that the model class can usually be represented in XML, whereas the view and the controller are more problematic. This has been addressed with the flexibility concerning "mixed" XML/Java Reference Frame specification techniques. The latter combine the simplicity of the document centered approaches with the expressiveness of the program centered techniques.

The implementation of the advanced XML based definition options for Reference Frames is relatively new (compared to the program centered approach). In addition, the usages of the system have to a large extent involved modeling languages with a rich operational semantics, which is the weak point of the document centered approach. Due to these two reasons, the XML based (and mixed) approaches for Reference Frame specifications have not been used widely up to now - this should be considered reading the corresponding evaluation parts in chapter 8.

7.1.2 Plug-In Management of Reference Frames

This subsection shows how the runtime management of Reference Frames, specified using any of the mechanisms defined in the previous subsection, can be done (and is implemented in the Cool Modes environment). Apart from system-internal representation issues, there are two design questions to address, both having a dynamic and a static solution.

- The first question is how Reference Frames are *offered* to the user: are selection and deselection (thus, a dynamic use) possible, or are all Reference Frames available all the time?
- The second question is related to the internal system design: is dynamic (i.e., runtime) *retrieval* of Reference Frames possible, or does the environment rely on a static list of Reference Frames used?

In Cool Modes, the dynamic option is chosen to address both questions, which motivates the name of "Plug-Ins" for Reference Frames. For the first question, a static solution would indeed by strange: the framework supporting a variety of heterogeneous modeling languages, a permanent (and enforced) visibility of all these languages is not likely to support users in their tasks. This is in particular true for the case of applications in learning scenarios, where the reduction of means may even have concrete educational reasons. In Cool Modes, the dynamic adding and removing of Reference Frames can be controlled through menus and dialogs - figure 7.4 shows the central dialog, which allows the adding of Reference Frames.

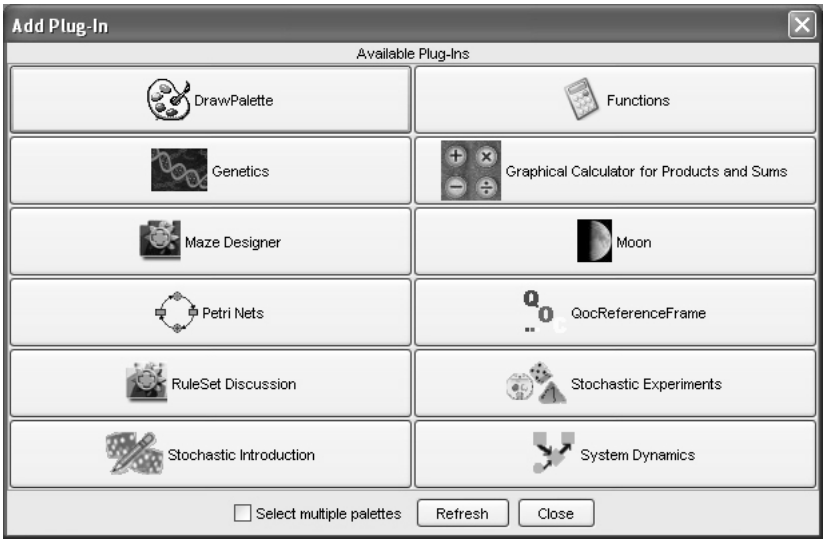

Figure 7.4: The Plug-In dialog of Cool Modes

As visible, the dialog contains the names and icons of the found Reference Frames. The retrieval of the Reference Frames is also done dynamically behind the scenes: upon a request to search for Plug-Ins, the `FrameController` initiates a search for all `.frame` files and all Java class files in the installation folder (recursively), including those in jar archives. A parser tries to parameterize template classes with the found `.frame` files, and (if successful) the `ReferenceFrameFactory` returns a `ReferenceFrame` implementation. For the class files, the Java Reflection mechanism is used to determine if the class implements the `ReferenceFrame` interface. Figure 7.5 shows the Plug-In search algorithm in a sequence diagram, and the corresponding class diagram 7.1 shows the classes important for this algorithm within their general architecture context.

As both the XML parsing and the Reflection mechanisms are relatively slow, Cool Modes remembers the Plug-Ins that a user has added, and offers shortcut options to add these again in future system usages. In addition, the Plug-In dialog is capable of storing the "found" results, so that not every time a user wants to add a Plug-In, the whole search process has to be started (yet, it is possible using the "refresh" button visible in figure 7.4).

7.2 Visual Interfaces and Interaction Paradigms

The description of Reference Frames in the previous chapters of this thesis already included remarks about foreseen user interfaces (e.g., in the interface shown in figure 6.1). Obviously, the user interface design is an important issue for the targeted aim of supporting collaborative modeling - users interact with the system (and thus, with the model they construct) through the functions and with the metaphors that the interface offers to them - in addition, the representations can have feedthrough functions and thus contribute to the collaboration process. Consequently, the design of the user interface is an important success factor of a collaborative modeling environment.

The aim of this thesis, however, is not to investigate which representation and interaction principles embodied in user interfaces for collaborative modeling are of advantage. For this reason, the Reference Frame implementation proposed in

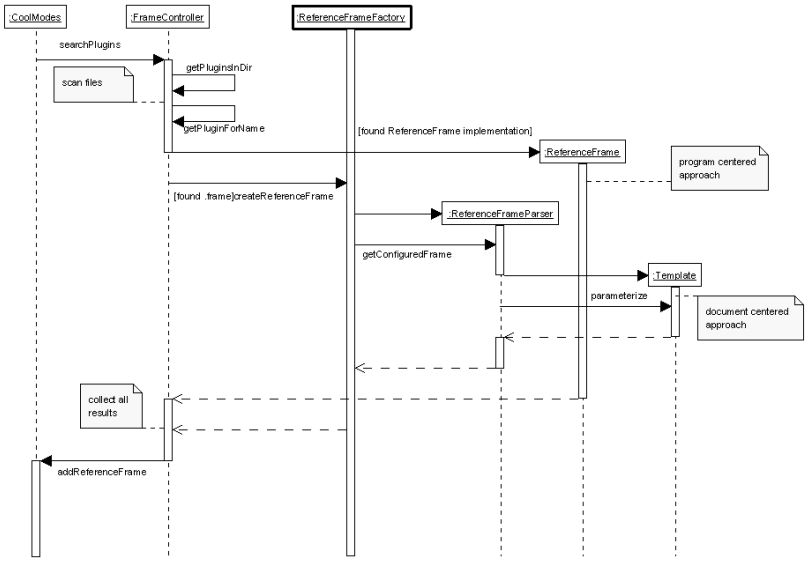

Figure 7.5: The Plug-In search algorithm of Cool Modes

chapter 6 is independent of concrete user interfaces. The latter are encapsulated and decoupled from the "rest" (the modeling language), so that certain user interface design decisions like, e.g., the way the user interface allows the user to access the node and edge types specified by the Reference Frame, can be made independent of the Reference Frame they relate to. Of course, "tailored" user interfaces for specific Reference Frames are not prevented with this approach, as a Reference Frame still defines "its" user interface.

The two following subsections outline how the Cool Modes implementation of user interfaces for modeling with Reference Frames is done.

7.2.1 Palettes: User Interfaces of Reference Frames

As contained in the `ReferenceFrame` interface shown in figure 6.1, user interfaces of Reference Frames are denoted with the term *Palette*. Figure 7.1 shows how the corresponding Java interface is embedded in the architecture. The most important part of this interface is a `getUI()` method, which returns a `javax.swing.JComponent`. This reflects the flexibility that the developer has in the design: the major part of figure 7.6 is the `JComponent` returned by this method. Typically, a `Palette` implementation serves the following purposes:

- It provides the user with a means to access the primitives (node and edge types) of the Reference Frame, and to build model graphs with these primitives.
- Apart from that, a Palette can contain control elements, which might be needed for, e.g., running simulations of models.
- The Palette can be the location that offers connections to external elements (e.g., devices or file types) and allows the user to integrate these (cf. examples in section 8.1).

Figure 7.6 shows an example of a Palette in Cool Modes. This example, taken from (Bollen et al., 2002), is in the field of System Dynamics - here, the user

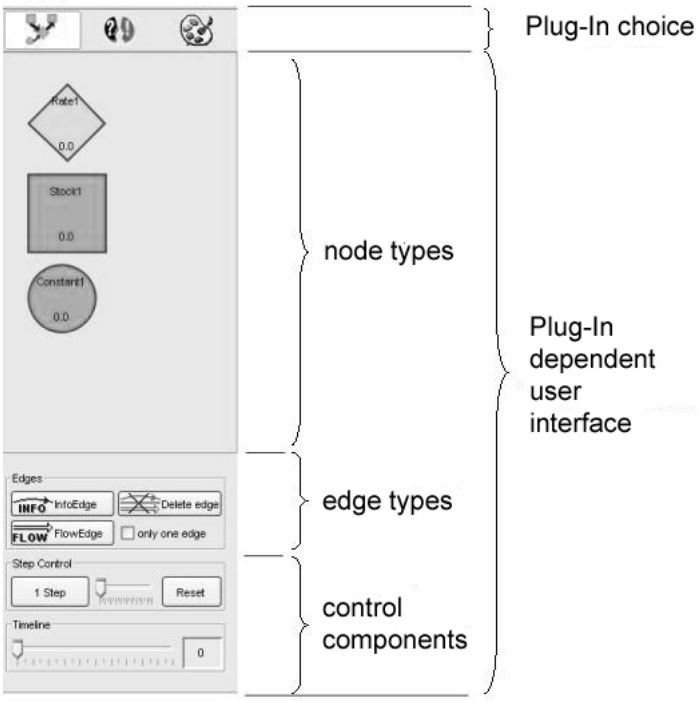

Figure 7.6: Typical Palette design in Cool Modes

interface of the Reference Frame contains three node types and two edge types, and in addition several control components which allow the step-wise simulation of models.

The figure also illustrates how multiple Palettes are managed: the top component of the frame contains buttons for all the Reference Frames that the user has activated (using the dynamic mechanism described in subsection 7.1.2), with icons on the buttons symbolizing the Reference Frames. In the example situation shown in the figure, two other Reference Frames (one for discussion support, and another for handwritten annotations) are active. The remaining (majority of) space in the frame is left to the UI component as defined by the Palette of the currently selected Reference Frame.

Figure 7.6 illustrates a frequently chosen design principle for Palettes as it has emerged during the usage of Cool Modes in several application areas (cf. section 8.1). Here, the Palette contains instances of the node and edge types that the Reference Frame offers. Using drag&drop (in a copy&paste metaphor rather than a cut&paste one), the user can create copies of these node instances in workspaces (cf. next subsection). The edge types are made available with a similar mechanism: upon activating a corresponding button, an edge of the selected type can be "dragged" in a workspace.

These principles allow the construction of models based on primitives offered by the Palettes, and imported into the workspaces from the Palettes through explicit and intuitive direct manipulation actions.

For the document centered specifications of Reference Frames (cf. subsection 7.1.1), user interfaces of the described style can be (and need to be!) automatically generated by the framework system. This is illustrated in figure 7.1: here, the `CustomReferenceFrame` class makes use of a `CustomPalette`, the latter being a parameterizable template for user interfaces of the type outlined above. The system

architecture shows that other general Palette templates can easily be integrated by different `ChangeableReferenceFrame` implementations, referred to from the XML Reference Frame specifications.

7.2.2 Workspaces

Palettes are the visual representation of *modeling languages* in the Cool Modes system. They offer to the user the primitives with which graph based models can be built. For the practical *construction of models*, additional user interface components are needed. In the case of Cool Modes, these model construction places are called workspaces.

Cool Modes allows the creation of multiple workspaces, which can be arranged freely on the desktop that the system offers (cf. figure 7.8). The basic function of a workspace is to allow the direct manipulation of the contained graph based model. Of course, the allowed actions on nodes and edges depend on their type - the framework system, however, offers generic drag&drop support within and across workspaces, connectivity features (i.e., the construction of graph structures), and some additional features like duplication of elements.

Obviously, the choice of a workspace orientation (versus other imaginable formats like, e.g., a page model, or a tree structure) to allow for multiple model graphs synchronously, is independent of both the Reference Frame approach and the Palette concept. There are indeed other applications that make use of the Reference Frame concept and its implementation including the Palette concept, but rely on a page orientation (Gaßner, 2003).

The advantage of a classic workspace model with multiple workspaces per desktop is its flexibility in terms of fine-granular control of synchronized and private elements: it is possible to maintain private workspaces together with shared workspaces on the screen, and thus (using simple drag&drop operations) "publish" results to the user group, or to develop private "test" solutions while watching what the rest of the group is doing in the shared space. The next subsection describes the options that Cool Modes offers (based on the Reference Frame approach and its implementation) in this respect.

As put, workspaces are the primary means of model co-construction in Cool Modes. In this application, a workspace consists of several layers: a background image layer, a static and a dynamic graph layer, and multiple layers for handwritten annotations. The graph in the static graph layer cannot be modified - thus, it can serve as, e.g., a "task description" component.

Workspaces are scrollable, can be saved (independently from the main "save" function of the system), and have auto-layout features for graphs. The embedding framework offers the following "standard" functions:

- load/save and export of models in various formats (e.g., SVG and JPG)
- clipboard functions: cut, copy, paste, and redo/undo (Jansen, 2003),
- workspace and Plug-In management,
- replay functions, and
- system-wide language management functions

Apart from these basic functions, Cool Modes includes connections to document archives and model checking functions (cf. chapter 9), and a number of features required for the support of synchronous cooperation via shared workspaces. These are described in the next section.

7.3 Collaborative Modeling Support

As discussed in the introduction of this thesis, not only the approach for *modeling* with heterogeneous graph based representations and its implementation are important issues in this thesis, but also the system-side support for *collaboration* is an explicitly targeted aim. On the conceptual and architectural level, this feature has already been addressed in the sections 4.4 and 6.3.1. Furthermore, the sections 5.3 and 5.4 contain a discussion of the chosen background technologies.

This section builds upon these results and demonstrates how cooperation support mechanisms can be implemented for the specific case of heterogeneous graph models contained in workspaces.

7.3.1 Shared Workspaces

Similar to a number of other environments presented in section 3.2, the collaboration support in Cool Modes relies on the principle of *shared workspaces*. Changes caused by a user in a shared workspace are propagated to the corresponding workspaces in the coupled applications. This leads to the conceptual *group interface* of a "shared graph space" with partial WYSIWIS usage metaphor. The WYSIWIS principle is not completely enforced with respect to scrolling (i.e., different views on workspace parts are supported) and positions of workspaces on the desktop. Furthermore, the flexibility in terms of *partial* synchronization (cf. next subsection) is at the same time a reduction in terms of WYSIWIS.

As presented in subsection 5.4.10, MatchMaker is the synchronization backbone of the COLLIDE JGraph library, so that its use also on the workspace and application level of Cool Modes makes sense. Indeed, the synchronization tree model that MatchMaker offers, together with the concrete tree instances created by the JGraph library, allows for a relatively straightforward approach for synchronizing Cool Modes workspaces via MatchMaker. Figure 7.7, a synchronization tree for Cool Modes, illustrates this. In the figure, two workspaces are synchronized. The synchronization tree consists of a root vertex, a first child (label //4), and one child tree per workspace (labels //4/5 and //4/16). The structure of a workspace subtree is determined by the layers that this workspace contains, plus one vertex for awareness information (see below). The highlighted part of figure 7.7 shows that the subtrees representing the graph layers are the same as those for the JGraph (cf. figure 5.5) - a consequence of the MatchMaker synchronization functions of the JGraph library used in Cool Modes workspaces.

Beyond the technical means for workspace synchronization, Cool Modes demonstrates the suitability of the Reference Frame approach for collaborative usage scenarios with a number of additional cooperation support features, such as the availability of flexible synchronization *modes* and the support for different modeling *phases* (cf. next subsections). With these degrees of flexibility, *workspace awareness* in an important issue. This is addressed in Cool Modes with several functions suggested by Gutwin and Greenberg (2004):

- Distinguishable *replicated mouse pointers* which visualize the mouse cursor positions of the different users in the workspace.
- An overview frame for scrollable workspaces (cf. lower right corner in figure 7.8), in which remote actions are symbolized by colors that represent the users who caused the action. This way, users can quickly locate activity regions of their collaborators (similar to a radar view).
- Plug-In dependent *feedthrough* mechanisms. E.g., the Petri Net Plug-In visualizes the firing of transitions so that collaborators can see the cause of

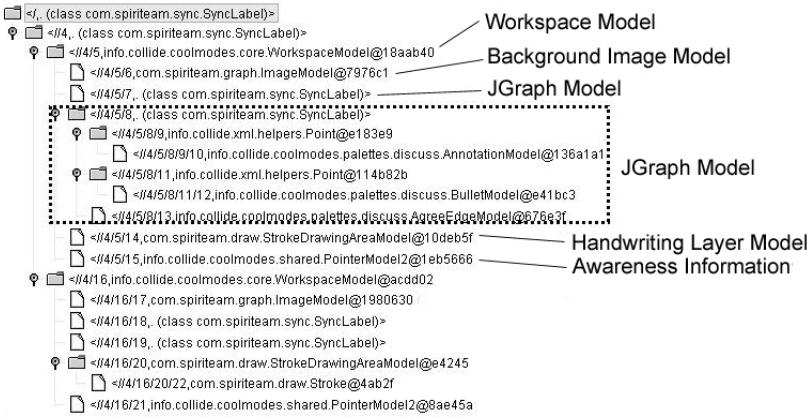

Figure 7.7: A synchronization tree for Cool Modes

changes in place states.

- A common *telepointer* (to enable the co-users to share a focal point in the workspace) can be implemented as a Plug-In, which provides a single node (e.g., in form of cross hairs) that can be dragged into a workspace from the Palette (and subsequently disappears in the Palette).

7.3.2 Synchronization Modes

Flexibility in terms of usage scenarios for the collaborative modeling framework is an important issue, especially under the premise that the tool is only part of a (learning) environment (cf. section 1.5), and therefore needs to be adaptable to various intended scenarios. In particular, multiple *cooperation modes* can be a key factor to support a variety of group usage scenarios. The synchronization functions available in Cool Modes are based on the shared workspace metaphor and have been motivated by the axiom not to restrict the educational designer in the collaborative settings he can orchestrate with the tool. Yet, this flexibility can cause both undesired inconsistencies (cf. figure 4.3), and needs for visualization (awareness) techniques to feed back to the users the synchronization state of the workspaces.

As a result of the trade-offs between flexibility and consistency, Cool Modes offers five different synchronization modes, which can be characterized simply through their primitive of synchronization:

Coupling by Application. Here, the complete Cool Modes application (i.e., all workspaces), are synchronized. Private actions are not possible within the system - instead, each user has a complete view on what the others are currently doing.

Coupling by Workspace. This synchronization mode offers private workspaces and shared workspaces synchronously, which allows users to, e.g., work on "test models", and publish them to the group after having verified them.

Coupling by Layer. Here, a workspace can be partially synchronized in the sense that some layers are shared, while others are private. The layer-based synchronization mode allows for, e.g., personal handwritten annotations attached to jointly used models.

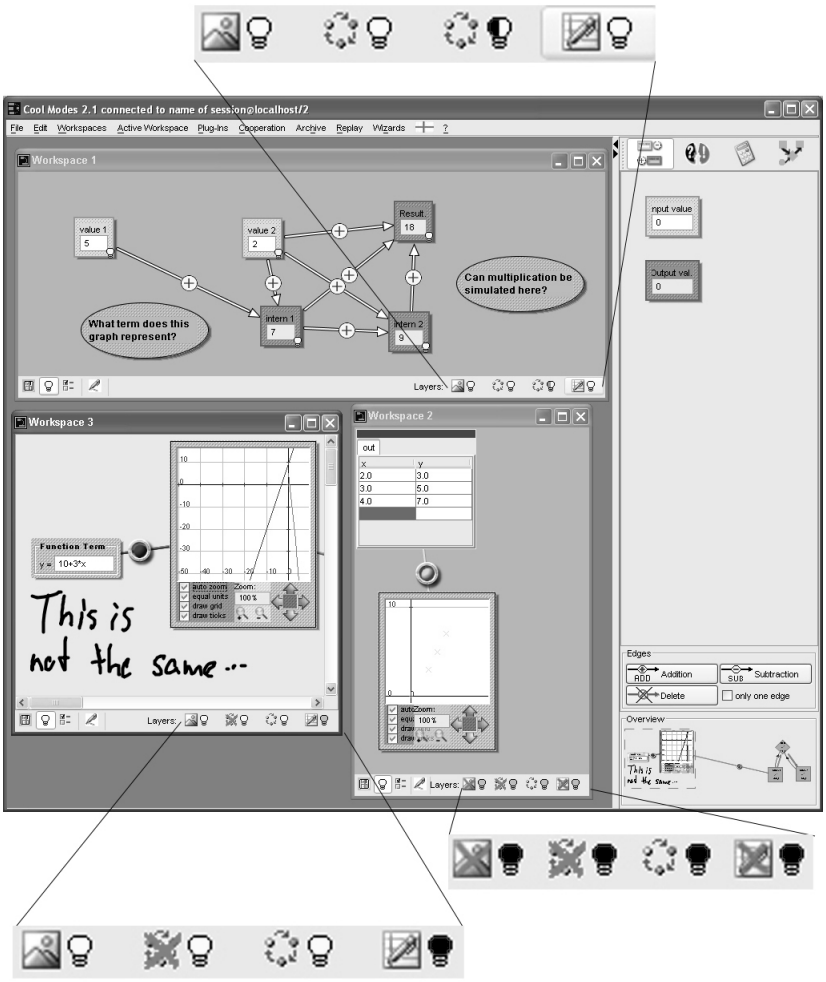

Figure 7.8: Cool Modes partially synchronized

Coupling by Reference Frame. This synchronization mode is based on user-selected Reference Frames, whose primitives are automatically "published" in the workspace. This allows a pre-selection of private and shared entities, e.g., to a priori distinguish comment elements from model elements.

Coupling by Synchronization Context. This "finest" level of synchronization allows the user to specify the nodes he wants to share. For reasons of consistency, not only these nodes are shared, but also (recursively) their synchronization context as delivered by the Reference Frame which defines the node to be coupled.

The last two points in the list are interwoven and can be combined, which ensures consistency for the Reference Frame based strategy even if synchronization contexts specify a larger scope, and allows for defining "synchronized modeling languages", plus single shared elements from other languages.

Figure 7.8 shows a typical partial synchronization scenario in Cool Modes, and outlines the visualization of synchronization as fed back to the users. The latter is done throughout the system using the metaphor of a *light bulb* which can be lit

(representing synchronization) or not (in the enlarged parts of the figure, the real colors have been replaced by black and white to highlight the differences).

In the figure, three layers of *workspace 1* are synchronized. This is shown by the three lit bulbs in the tool bar of the workspace. The graph layer is synchronized by Reference Frame: here, the elements of the calculation net are shared, but not the comments. In the user interface, this is represented through lit bulbs attached to the shared nodes, and a semi-lit bulb in the tool bar. *Workspace 2* is prepared for sharing (including available awareness information), visualized by lit bulb in the left part of the tool bar. Yet, none of the layers is shared. This is reflected through the non-lit bulbs in the right part of the tool bar. Finally, *workspace 3* has three synchronized layers and one private handwriting layer, the latter visualized with the non-lit bulb next to the handwriting symbol in the tool bar. Furthermore, the user has locally hidden one (shared) layer, which is symbolized with the crossed layer icon. The figure shows the dilemma that growing flexibility concerning coupling options will typically go in line with more complexity with respect to the synchronization awareness mechanisms - though, of course, there might be more suitable representations than the light bulb metaphor.

7.3.3 Modeling Phases

The previous two subsections outlined how the support for collaboration in COOL MODES is implemented based on shared workspace principles. However, the support for collaborative *modeling* (as opposed to *general* collaborative tasks) can be more specific in that modeling activities typically involve different phases, which may correspond to either different representations, different tools, or different usage modes like exploration or simulation (Löhner et al., 2003; Ainsworth, 1999; Kurtz dos Santos & Ogborn, 1994).

If phases in a collaborative modeling task can be associated to different modeling languages, then the Plug-In concept for Reference Frames is a suitable support for these phases, as representational notations (e.g., formal modeling languages like System Dynamics, more exploratory notations like causal feedback loops, or argumentation supporting languages) can be dynamically used. To contribute to supporting these phases in the *collaborative* modeling process, COOL MODES optionally offers the propagation of adding or removing Plug-Ins (via MATCHMAKER) to synchronized instances. This way, a common group phase can be maintained - yet, also independency in terms of used Plug-Ins is possible, thereby enabling scenarios in which, e.g., one user builds a formal model, and others have notational primitives (like question symbols, etc.) suitable for challenging the model.

Modeling can be seen as an activity which contains both constructive phases in which models are built (or revised), and testing phases in which the created models are tested and "run" in simulations. Therefore, apart from phases that are related to modeling languages, it is also reasonable to distinguish between phases corresponding to usage modes.

Typically, a user interface design based on the direct manipulation metaphor (Shneiderman, 1983) would avoid the concept of "usage modes", but instead try to build "modeless" integrated solutions in which the edits of a user have direct impacts on the behavior or the model. However, *collaborative* modeling situations might have different requirements: if one user intends to edit a model, while concurrently another one tries to conduct test runs on the model, this may lead to confusion and unintended inconsistency.

To offer a means of coordination here, the Cool Modes environment allows the selection of a an *interaction mode*, which is a synchronized system state independent of Reference Frames and workspaces. The framework system controls this state and delegates state changes to the single Reference Frames - methods to react upon

state changes are foreseen in the `ReferenceFrame` interface. The list of modes is not closed - however, three modes are currently foreseen:

- An *integrated* mode, which allows both editing of model graphs and simulations of models. This is the default system mode.
- An *edit* mode, which lets the users construct and revise models, but does not support any simulations.
- A *simulation* mode in which the model structure is fixed (i.e., edits are impossible), and simulations can be conducted.

Using the `switchMode(int mode)` method of the `ReferenceFrame` interface (cf. figure 6.1), all Reference Frames are notified about interaction mode changes. Each Reference Frame can then cause the necessary changes. Here, the `mode` parameter is a constant defined in the `ReferenceFrame` interface. The changes are typically located in the Palette of the Reference Frame (e.g., enabling or disabling of control elements), and in the models in the workspaces (e.g., enabling or disabling text input fields). The latter is supported by the Model View Controller separation in the JGraph library: typically, only the *view* component of nodes will have to be changed corresponding to the interaction mode.

As described, major parts of the implementation of the phase support based on interaction modes is left to the Reference Frames: the framework system, however, offers the *option* for interaction modes and ensures their *consistency* and *propagation*. Obviously, not all interaction modes are reasonable for all Reference Frames. Typically, "informal" languages with a low degree of formal semantics, e.g. to support discussion and argumentation, will not offer a specific "simulation mode" and therefore ignore interaction mode changes.

7.4 Interoperability Issues and Design Aims Met

This chapter presented the Cool Modes environment as one possible implementation on top of the Reference Frame architecture presented in chapter 6. Cool Modes relies on the principle of shared workspaces and emphasizes flexibility with respect to the primitives of sharing, thereby illustrating the options that the Reference Frame foundations and the proposed system architecture offer. Furthermore, the system contains some support mechanisms specifically related to phases which may occur in collaborative modeling activities. These mechanisms are lightweight in the sense that Cool Modes offers only basic technical process support. Further mechanisms, which would, e.g., address phase scripts or analysis and recommendation techniques for specific situations in collaborative modeling processes, are beyond the scope of this thesis.

As an effect of the conceptual split between abstract architecture and concrete system design, the Reference Frame approach is not bound to the Cool Modes environment: the FreeStyler application (Gaßner, 2003) is an example of an environment which uses the Reference Frame architecture, but provides a differently designed (page based) user interface and further functionalities that are targeted towards knowledge construction and management rather than flexibility in synchronous sharing mechanisms.

Revisiting the list of challenges and aims for this thesis as developed in section 1.5, the criteria that were not yet addressed in chapters 4 or 6 can be characterized as being (at least partially) dependent on concrete system implementations. As argued, the Reference Frame approach and architecture are independent of concrete applications. Therefore, the fulfilment of the criteria for the specific case of

COOL MODES can usually not be transferred to other imaginable implementations of Reference Frames.

The criterion of graphical user interfaces and views for model definition and integration (Dolk & Kottemann, 1993) (cf. criteria list on page 19) is addressed with the flexible Palette concept. As both Palettes and Reference Frames are Java classes, the integration of translator components that convert external data formats as demanded by Dolk and Kottemann (1993) and Roschelle et al. (1999) (cf. page 20) is also generically supported: the translators can simply be embedded into these classes or delegates. The examples in section 8.1 illustrate the flexibility that COOL MODES offers in this respect.

The advanced persistence mechanisms for models and modeling languages demanded by Roschelle et al. (2000) (page 20) are addressed with the XML serialization not only of models, but also (partially) of modeling languages (cf. subsection 7.1.1). The dynamic Plug-In mechanism is both easy to use (cf. figure 7.4), and provides for dynamic runtime interoperability in the sense of Roschelle et al. (2000).

Task and social interoperability issues are met basically through the Plug-In approach and the collaboration support features. Of course, an application framework can not generically *offer* these kinds of high-level interoperability - the concrete usage context (including tool, task and social factors that are beyond computer control) has an important impact, as expressed in the discussion in section 1.5. Following this argumentation, the role of a framework in supporting social and task interoperability can only be defined through the functions and degrees of flexibility that the tool *offers* to the users in order to *allow for* an interoperability in these senses.

For the case of COOL MODES, *task interoperability* is addressed primarily through the provision of a co-constructive environment with shared workspaces and appropriate awareness mechanisms. Understanding modeling languages as tools, tool switches are avoided through the dynamic and interactive mechanisms for "plugging in" languages. In addition, phases in the collaborative modeling tasks are supported by synchronization functions for Reference Frames and application-wide "interaction modes", which reflect the current state of the modeling process.

From the viewpoint of *social interoperability*, the primary characteristic property of COOL MODES is that it allows the integrated collaborative use of different tools. In particular, it is possible to switch tools *without leaving the group work context*, i.e., the collaboration session. For these sessions, flexible parametrization options exist (cf. subsection 7.3.1). This degree of flexibility was chosen in order to allow the orchestration of various social forms with the collaborative modeling environment COOL MODES.

Chapter 8

Applications and Evaluation

The previous four chapters of this thesis presented the Reference Frame approach from both conceptual and implementation points of view. This approach addresses the target specifications discussed in the introduction, in particular in section 1.5.

The fulfillment of some "success criteria" has already been discussed in various parts of this thesis, in particular in sections 4.7, 6.5, and 7.4. The argumentations in these sections were (mostly) on a theoretical level, in accordance to the formal and implementation oriented character of the chapters. This goes well with most parts of the criteria listed in section 1.5, in particular with those associated to formal or technical system requirements (e.g., syntactic and semantic interoperability across visual languages, or the criteria identified by Dolk and Kottemann (1993) (see page 19 of this thesis)). In fact, most of the core functional criteria (like the support for heterogeneous model structures, or the retainment of a shared semantics in partially shared models) were proved to be fulfilled.

However, the criteria list discussed in section 1.5 does not only consist of requirements which can be formally proved (or at least justified). Indeed, the list is quite heterogeneous with respect to required methodological approaches for verification or substantiation. For some more usage oriented aspects, including the ones related to the educational suitability of the system rather than its core functionality, and those focusing on reusability, empirical techniques are more appropriate.

For two reasons, the evaluation of these "soft criteria" is difficult. First, the Reference Frame approach and its Cool Modes implementation has a framework character. Using the system makes sense with Plug-Ins only, which in turn have a strong impact on the conception of the application framework by the users. Thus, any study would be dependent on the Plug-Ins, which would lead to a high number of required studies to eliminate this effect. Apart from this general framework evaluation issue, the second problem is the intention of the system: as argued in section 1.5, Cool Modes is *not* designed to be a learning environment of own right, but primarily as a tool that can be flexibly used *within* learning environments (or: contexts). This would not be considered by any evaluation study which, employing e.g. a classical pre-test/post-test design, measures "learning success" on some scale.

To address these problems, the evaluation parts in the following of this chapter base on interviews conducted with several teachers who have used the Cool Modes system in their regular classes, and with programmers who have developed a number of Plug-Ins. Due to the amount of experience that the interviewed persons (both programmers and teachers) have with the system, a certain degree of Plug-In independency can be reached in the integrated view on the interviews.

The analysis of these interviews gives answers to the questions related to *usability*, *task* and *social interoperability* (teacher interviews), and *expressiveness*, *ease of use* and *reusability* (programmer interviews).

To furthermore outline the *flexibility* of the system not only from a theoretical perspective (as done in sections 6.5 and 7.4) but also based on concrete implementation results, the next section presents some example application areas of Cool Modes. The diversity of these examples also indirectly substantiates the general *usability* of the framework system: major problems in this area would surely have prevented repeated and successful usage.

8.1 Application Areas

The Cool Modes system presented in chapter 7 has a development history of (roughly) four years: a first version of the system was released in 2001. At the time of this writing, 41 differently targeted Plug-Ins are available in the software archives of the COLLIDE research group, external ones do exist additionally.

During the previous nine months (i.e., since the last major release), more than 400 users downloaded the Cool Modes framework from the "official" place in the web portal of the COLLIDE research group. The downloads of Plug-Ins sum to more than 1400 during the same period of time. All these numbers do not consider the usage of the system in specific contexts (e.g., within research projects): here, other (uncontrolled) distribution mechanisms have been used, so that the real number of downloads is significantly higher than the presented numbers.

The statistics show that a complete listing of the application areas of Cool Modes is not realistic within this thesis. To give an overview of the diversity of application areas for Cool Modes as reached through its Plug-In concept, the following subsections shortly illustrate some use cases and different *roles* of the Cool Modes system in selected recent research project contexts, and the associated different *requirements* that were imposed on the system. In addition, some "extreme" cases of Plug-Ins are shown in subsection 8.1.4.

8.1.1 SEED

The SEED ("seeding cultural change in the school system through the generation of communities engaged in integrated educational and technological innovation") research project (SEED project homepage, n.d.), supported by the European Union, lasted from 2001 until 2004. An important aim of this project was the generation of integrated communities, consisting of teachers, researchers, and software developers, with the purpose of promoting innovative educational practices which are possible through pedagogical and technological innovation. The general reason for building integrated communities was to combine educational and technological expertise in designing, developing and implementing innovative activities.

The local community established at Duisburg and maintained through and beyond the lifetime of SEED included a number of interested local teachers. Evolving from this, a number of regular school lessons (and lesson series) which integrated Cool Modes and the related FreeStyler application (cf. section 7.4) have been conducted. All the teachers that volunteered as interview partners (cf. section 8.2) also participated in this SEED community.

Based on this project embedment, the role of the Cool Modes system in the SEED scenarios is closely related to the original motivation for the system discussed in chapter 1: *embedded in realistic school usage contexts, the system is used as a means to enable or support collaborative modeling tasks.* Two immediate consequences of this are the requirement of *technical reliability* (in regular school lessons of 45 minutes duration, the tolerance of teachers against buggy software or low performance is relatively low), and *usability*: in most cases, a regular classroom use does not leave much time for lengthy explanations of tool use, so that an intuitive

Figure 8.1: Cool Modes in classroom use: the stochastics scenario

usage is important. The success of the Cool Modes usage in these scenarios shows that these criteria were at least basically met (cf. section 8.2 for details).

Concrete Plug-Ins developed and/or used within this SEED context include the System Dynamics implementation (Bollen et al., 2002), a Stochastics environment (Lingnau et al., 2003), computer science Plug-Ins (UML and finite automata), a biology environment in the field of genetics, and a Plug-In designed for knowledge management that was used in a German lesson - see Gaßner (2003) for a detailed description of the usage situation.

Kuhn et al. (2004) have presented the educational setting employed in the "stochastics" lessons, including an evaluation based on a number of lesson sequences. Figure 8.1, taken from this paper, shows two different usage modes of Cool Modes in the lesson series. The figure underlines the approach of "seamless media integration in the classroom", fostered by the SEED project, and also exemplifies the approach of computer based tools being integrated parts of learning environments as presented by Hoppe (2002).

8.1.2 COLDEX

COLDEX ("Collaborative Learning and Distributed Experimentation") is a research project funded by the European Union from 2001 to 2005 (COLDEX project homepage, n.d.). In this project, technological approaches and computational tools to foster scientific experimentation, modeling and simulation in distributed collaborative settings have been developed and used to establish an intercultural (European-Latin American) community of learners.

In this attempt, the COLDEX project focused on educational scenarios which involve the study of visual and other perceptual phenomena, including astronomical and seismic measurements, from both scientific and subjective experiential perspectives. The target groups of COLDEX ranged from higher secondary education to academic beginners.

The pedagogical idea of "challenge based learning", adopted within COLDEX, includes open-ended learning activities which confront students with realistic (and complex!) scientific challenges. "Digital experimentation toolkits" (DEXTs), containing both virtual and physical tools, are handed to students and allow them to undertake investigations.

In a number of these DEXTs, Cool Modes with different Plug-Ins had the role of the (virtual) modeling tool. Different from the SEED context, here the *connection of the system to real physical devices* (including data transfer or exchange) was an essential factor. Of course, these usages demanded *flexible system interfaces*, and a *high expressiveness* of the Plug-In concept.

Within this COLDEX context, three Plug-Ins have been developed and used with Cool Modes:

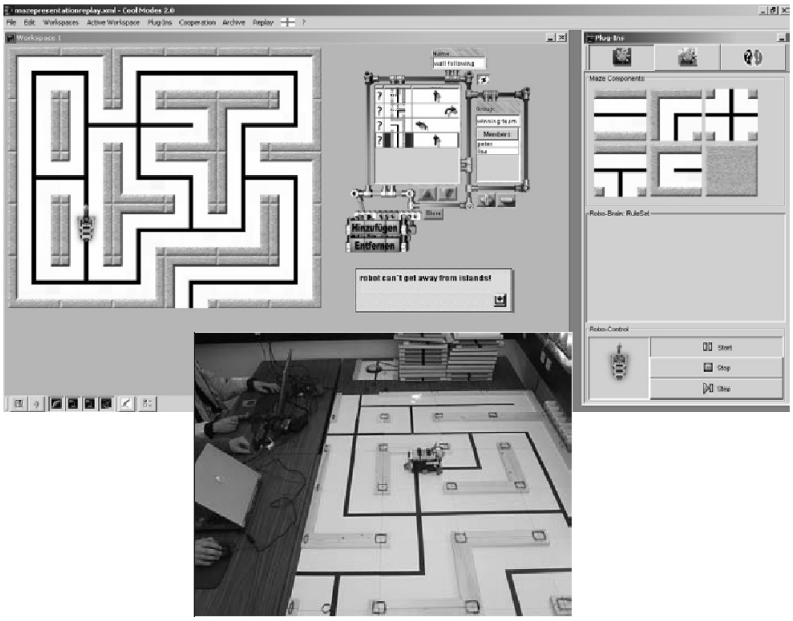

Figure 8.2: Cool Modes in the COLDEX maze scenario

Astronomy. The astronomy scenario, presented by Hoeksema et al. (2004), deals with space objects like the moon or planets. With real physical telescopes, images of these objects can be taken. Two Cool Modes Plug-Ins can then be used for image processing and calculating heights of craters based on distance measurements in the images and geometric rules.

Seismology. This scenario, presented by Baloian et al. (2004), is about the investigation of the geological phenomenon of earthquakes. Using real data gathered from seismic measurement stations, students can calculate epicenters and hypocenters of earthquakes. The calculation process is facilitated through a specifically designed Plug-In. The scenario possible with this Plug-In allows supporting scientific inquiry learning well, since it includes challenging problems, real data, and options for investigations and modeling.

Maze. In this scenario (Jansen, Oelinger, Hoeksema, & Hoppe, 2004), the problem is to help a robot out of a maze through case based rules in a "reactive programming" approach, which defines global robot behavior based on single rules that can be provided "on demand". The COLDEX implementation involves a physical maze with Lego Mindstorms robots as well as a Cool Modes implementation for the virtual parts (cf. figure 8.2). Rules can be transferred from the virtual environment to the physical robot.

8.1.3 "Shared Workspaces"

The "Shared Workspaces" ("Co-constructive working and learning in replicated shared workspace environments: Visual languages, automatic analysis, and support") project was funded by the German Science Foundation from 2000 until 2004. This project was part of a research program on networked knowledge communication in groups (Wissenskommunikation project homepage, n.d.).

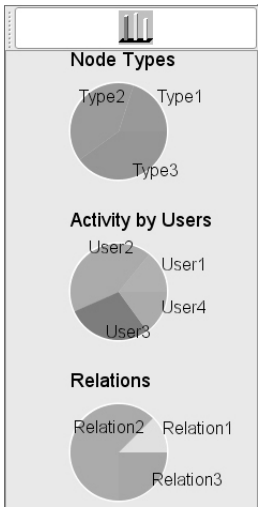

Figure 8.3: Palette of the feedback charts Plug-In

Compared to the previously reported projects, the "Shared Workspaces" project was less application oriented. Using rather analytical and theoretical approaches than "field studies" with realistic user scenarios, it primarily aimed at advancing the state of art in CSCL and knowledge communication with respect to semantic enrichment and support in CSCL environments as well as computerized analysis of cooperative activities.

Particular results of the "Shared Workspaces" project are in the fields of state based and action based interaction and collaboration analysis methods (Gaßner, Jansen, Harrer, Herrmann, & Hoppe, 2003), and the use of real-time feedback on the collaboration process to support learning teams (Zumbach, Mühlenbrock, Jansen, Reimann, & Hoppe, 2002).

The Cool Modes environment has been used in the "Shared Workspaces" project primarily as a *system with which controlled lab studies were conducted*. The log files generated by the system (which have their original purpose in serving as replay files) served as a means for data analysis. This usage of the system included *technical reliability* and high *usability* as critical requirements, as a too high amount of "tool talk" caused, e.g., through technical problems, would have reduced the quality of the material available for analysis.

Concrete Plug-Ins used within these studies included simple discussion support languages as well as specifically designed "feedback" Plug-Ins, which provide statistics elements that, put in a workspace, mirror back interaction parameters to the users for self-reflection. Figure 8.3 shows the Palette of this Plug-In: here, three different analysis and feedback options (nodes by user, edges by type, and nodes by type) are visible - put in workspaces, the sample categories for user names and element types are replaced by the ones that concretely exist.

8.1.4 Plug-Ins for Non-Standard Applications

Almost all the Plug-Ins that have been developed for usage within Cool Modes (or FreeStyler) follow a pattern which is explicitly foreseen already in the conceptual notion of a Reference Frame, and even more in the system architecture and the basic interfaces: the essential and most important definition of a Plug-In is the set of node and edge types it offers to the users. The Palette is designed primarily as a space

which offers prototypical instances of the nodes and edges, and allows the user to drag copies of them into workspaces in order to build models (cf. section 7.2). However, there are a number of exceptional cases which do not follow this pattern, and are therefore useful to demonstrate the flexibility of the Plug-In concept. Three categories of these exceptional cases are briefly summarized in the following:

Analysis and Feedback. Already motivated in example 4.6 in subsection 4.6.2 and shortly illustrated in the previous subsection, Plug-Ins can have a "meta functionality" in that they are designed for acting upon structures created with other Plug-Ins. Here, one example is the "interaction feedback" application (Zumbach et al., 2002). Another one involves a domain independent "model checker" (Herrmann et al., 2003), which is implemented as a Plug-In and can be used to verify models against known solutions (cf. section 9.2).

Element Generation. Another cluster of "non-standard" Plug-Ins can be characterized through their ability of *generating elements* and automatically inserting them into workspaces according to some system-external event. The Palettes of these Plug-Ins do typically not contain any primitives for drag & drop. Here, examples include an "external devices" Plug-In, which is capable of generating text nodes and image nodes from the content of certain folders in the file system (this was intended for integrating digital cameras and scanner pens with text recognition), an SMS simulation Plug-In (Bollen, Eimler, & Hoppe, 2004) which puts short messages sent from mobile phone PDA emulation into a workspace, and also an "e-mail" Plug-In. The latter does not contain any user interface at all, but launches a local e-mail server and listens to incoming messages. Upon reception, a corresponding entry is generated in a workspace.

Communication Support. Based on the Reference Frame approach, the Plug-In concept of Cool Modes is originally intended to describe and encapsulate modeling languages to be used collaboratively. Yet, the expressiveness of the Plug-In notion has lead to some "abusive" forms of Plug-Ins which contain elements that are designed for pure *communication* support rather than as a means for collaborative modeling. These Plug-Ins have representational primitives that they offer to the users, but the primitives and are not intended for building models out of them. Here, examples include a chat Plug-In (which enables having multiple chat nodes in workspaces), and a "voting" Plug-In. In the latter, the Palette allows the construction of poll with, put in a workspace, enables the users to vote anonymously. Both Plug-Ins illustrate that even though Cool Modes does not come with advanced built-in communication features (apart from the shared models), such an extension is possible by means of the Plug-In concept.

8.2 Teacher's Views

During the last years, a number of teachers has used the Cool Modes application with different Plug-Ins in the context of experimental school lessons or even their regular teaching activities (cf. subsection 8.1.1). Even though in most cases no empirical evaluation of these lessons was conducted, the personal and subjective experiences of these teachers are an important source of information for evaluating the "soft goals" within this thesis (cf. introduction to this chapter).

For this reason, I conducted interviews with three teachers. These interviews were designed as "focused interviews". Merton and Kendall (1946) state that this semi-structured form is suitable for addressing situations with complex clusters of

Table 8.1: Guide for the teacher interviews

Topic	Subtopics
Usage situation	Infrastructure: computers and network in classroom Lessons conducted with Cool Modes: topics, classes Role of software in the lessons Other employed media: comparisons?
Inter-operability	Used Plug-Ins Were Plug-Ins used together? If yes: Which ones? Was usage problematic? If no: Reasons? When would this be reasonable?
Cooperation	Did students collaborate in the lesson? If yes: How? Was this intended? Role of Cool Modes? If no: Why not? When would this be reasonable?
Assessment	Did Cool Modes contribute to learning outcome? How? Teacher and Learner role: modified? Is Cool Modes flexible or more an "expert tool"? Usability issues General points of criticism and improvement suggestions

dependent factors - situations in which large series of successive experiments were needed for an ideal experimental design:

> *"The focused interview provides a useful near-substitute for such a series of experiments [...] Such a procedure provides an approximate solution for problems heretofore consigned to the realm of the unknown or the speculative."* (page 543)

The interview situations met the characteristics of focused interviews (personal involvement of interviewed persons, previous content analysis and initial hypothesis available, interview guide to locate the major areas of inquiry, and interview focused on the subjective experiences of the persons) and were conducted according to the recommendations given by Merton and Kendall (1946) and Liebold and Trinczek (2002). Table 8.1 shows the interview guide that was used as a loose structure for the interviews.

The analysis of the interviews was based on the methodology proposed by Liebold and Trinczek (2002), differing in that the coding was done on the audio files directly instead of using written transcripts. Table 8.2 contains the pre-determined codes used for this process.

The interviews were conducted in German, the native language of the interviewed persons and the interviewer. To be consistent with the text of this thesis, all quotations are translated to English.

8.2.1 Usage Context

The three interviewed teachers have different relations to the COLLIDE research group: teacher 1 is in loose contact with the research group and uses the Cool Modes system regularly as part of his lessons. Teacher 2, who had been a student assistant of the research group some years ago, is currently not affiliated to the group in any way, whereas teacher 3 has an intermediary position: he works part time at school and part time within the research group.

The three teachers have used the Cool Modes system in different contexts with respect to a number of dimensions. Table 8.3 lists the variations in terms of student age, taught subject, used Plug-Ins, and number of student groups.

Table 8.2: Coding categories for the teacher interviews

Code	Description
Context	Statements about the usage scenario for Cool Modes
Usability	Statements that express the quality of teacher's and student's experiences in interacting with the Cool Modes system.
Flexibility	Statements which express the teacher's conception of the possible usage fields of Cool Modes in educational scenarios.
Task interoperability	Statements that refer to task specific characteristics of the tool, e.g. related to modeling, task specific collaboration, or work phases
Social Interoperability	Statements which refer to Cool Modes used in groups including general remarks about collaborative usage.
Pedagogic assessment	Statements about the general usefulness of Cool Modes in educational situations.

Table 8.3: Teacher's usage contexts of the Cool Modes system

Teacher	Grade	Subject	Plug-Ins	Usage
1	12th	Biology (Ecology)	System Dynamics, Functions, Discussion	5 times
2	12th	Computer Science (Modeling)	Petri Nets, Handwriting	2 times
3	9th	Mathematics (Probability)	Stochastics, Discussion, Handwriting,	2 times
	11th	Computer Science (Modeling)	UML	1 time

In addition to these dimensions, also the technical infrastructure that the teachers had access to during the lessons varied: teacher 1 and 3 (working at the same school) had access to a well-equipped computer room with modern networked computers and a data projector, whereas teacher 2 only had old and slow machines (100 MHz) that did not allow for a use of the synchronization mode of Cool Modes.

A common point of these cases is that a group usage of Cool Modes in the sense of having two or three students per computer was employed. This was largely due to the fact that the number of students exceeded the number of available computers - however, the teachers reported quite positive on this (cf. subsections 8.2.4 and 8.2.5).

Apart from the details presented in the forthcoming subsections, the general impressions about the student groups as reported by the teachers differed considerably: teacher 1 reported on very interested students who felt "important" being in contact with university and research. Teacher 3 shared this positive attitude, whereas teacher 2 stated that his class consisted of a number of students that were only partially interested in his lessons.

8.2.2 Usability

The statements given by the teachers concerning the usability of the system can be classified in three subcategories:

1. General statements
2. Specific statements about certain Plug-Ins
3. Concrete problems

In general (i.e., speaking of the Cool Modes framework system independently of concrete Plug-Ins), the usability of the system was assessed as high by all teachers. Comparing Cool Modes to conventional system dynamics modeling tools like Stella or DynaSys, Teacher 1 stated:

> *"As a biology teacher I am happy to have a tool that is easy to use."*

Even with his limited experience with computers in general, he did not have any problems using the tool in the classroom context, and he did not report on major problems about his student's usage of Cool Modes, although *"some required a number of explanations."* Teacher 2 confirms this:

> *"I think this is fairly intuitive. No larger preparation time necessary. [...] That went without problems. One could directly start."*

The statements of teacher 3 are in the same direction. His introduction to the system in the classroom is nearly only about the Plug-In, which is only possible due to the intuitive usage of the framework system. In addition, he points out that the Plug-In character of the system makes further usages (with other Plug-Ins) easier, as the general interaction principles remain the same.

All three teachers evaluated the usability of the Plug-Ins that they used as very high. Teacher 1 also said that the students had no problems in using the different Plug-Ins together. The other teachers did not explicitly refer to this aspect, but their positive statements about the overall system have to be seen in the light of multiple Plug-Ins used in an integrated manner (cf. table 8.3). Teacher 3 also points out that the synchronization mechanism of Cool Modes can be handled by the students easily in the current system versions.

Table 8.4: Usability problems identified by the teachers

T.	Problem	State
1	File handling problems, loading impossible	Plug-In specific, solved
2	Font size and style not changeable	Plug-In specific, unsolved
1&2	Workspaces too small for large models	solved
2	No line break in text nodes	Plug-In specific, unsolved
3	Synchronization mechanisms difficult to use	solved
3	Printing functions are not flexible enough	unsolved
3	Replay mechanism too complicated to use	unsolved
3	Missing context help functions	unsolved

All teachers had specific points of criticism. These are listed in table 8.4. As these statements partially refer to older Cool Modes versions, the last column of the table indicates whether the mentioned aspects are still "open issues" or already dealt with.

The table shows that some of the mentioned points have already been addressed. For the important case of the workspace size, this has been done with scrollbars and the corresponding awareness mechanisms (cf. subsection 7.3.1). In accordance to the generally positive statements about the framework system, the (unsolved) problems reported by the teachers relate to specific detail functions, not to the core fundamentals (like, e.g., workspace or Plug-In management and handling, or collaboration support features) of the system.

8.2.3 Flexibility

Concerning the options and limits of using Cool Modes in school lessons, two topics emerged during the interviews:

1. Possible roles of the system in school lessons
2. Functional limits and options of the system as such

The former aspect was addressed by all teachers. Here, two different basic roles were mentioned: the use of the system as a medium for working on modeling tasks, and the possible use for presentations and discussions of outcomes (using a data projector to project the screen content). Going more into detail, teacher 3 added that Cool Modes offers support for a variety of different tasks related to modeling in education (cf. Milrad et al. (2002) for a theoretical perspective on this). He has used the system to:

- introduce and demonstrate the basic model primitives and their *usage*,
- work with existing models, *varying parameters* and *interpreting the changes* in the simulation outcome, and
- *structurally modify* or *create* models that meet a given problem description.

The aspect of functional limits and options (apart from the specific school usage contexts) of the system was also mentioned by all three teachers. Interestingly, one common point is that they all report on certain limits that the software had, and at

Table 8.5: Functional limits and the teacher's workarounds

Teacher	Problem	Solution
1	Quantitative data analysis components insufficient	Convert saved files to CSV format and use Excel
1	Design options for label nodes insufficient	Insert snapshots from Word into the workspace
2	No element for noting formal Petri Net semantics (e.g., marking tables) available	Use of handwriting
3	Missing text nodes in Stochastics Plug-In	Use of handwriting and Discussion Plug-In

the same time present ways to overcome these (cf. table 8.5). The flexibility that the system offers through its openness (Plug-In concept and integration of different media resource types as nodes in the graph) seemed to play an important role in their system usage, as it allowed for creatively solving problems caused by functional limitations. Talking about his experiences with the system, teacher 1 summarized this as follows:

> *"I went pretty far and saw the limits clearly. But you can help yourself. [...] The further functions are not so important. It does not make sense that you people have sore fingers programming functions that we have on our computers anyway for long. [...] It is legitimate to say that you do not reinvent the wheel a second time, but just mount it."*

8.2.4 Task Interoperability

The analysis of the interviews with respect to task interoperability yields a large number of statements. These can be classified into the following five subtopics:

1. Plug-In interoperability used for task purposes
2. Cross-tool interoperability
3. Modeling support in a narrower sense
4. Task specific collaboration issues
5. Phase support

Except for subtopics 2 (only teacher 1) and 4 (only teacher 2 and 3), all the points have been addressed by all three teachers.

Plug-In interoperability has already been dealt with in the previous subsection, in the context of the teachers classifying this as a means of flexibility which contributes to task support. Both the labeling of formal models with "informal" elements from other Plug-Ins and the option for handwritten annotations were mentioned here. However, teacher 3 added to this that the students did not frequently use the labeling of models with text pieces in his lessons. He identifies the lack of private access to the data as a possible reason for this: as the students had to share the computers, they preferred documenting their results on paper instead of using the digital tool only (and losing access to the data when they have to leave the computer). Concerning task support, teachers 1 and 3 also identified

the interoperability between data visualizer components and model components as essential factors of the system.

As already discussed in the previous subsection, teacher 1 made explicit use of the outward interfaces of Cool Modes in addition to the system-internal wiring options. In particular, he reports on the usage of data *exports* from the Cool Modes system dynamics model (in particular its simulation results) to Microsoft Excel for further data analysis and visualization (these exports were done by editing the saved Cool Modes XML files to achieve CSV format), and data *imports* from arbitrary tools into the workspaces using screenshots and importing the images into the graphs. The latter form of *cross-tool interoperability* was used to solve the perceived insufficiency concerning font sizes, colors, and styles.

All three teachers identify the dynamic modeling options that Cool Modes offers as a central feature which motivates the use of system in the lessons. Subsection 8.2.6 briefly discusses the pedagogic statements they make about this. Leaving out these educational dimensions, a summary of their statements is that Cool Modes suitably visualizes the complex model simulation processes (teachers 2 and 3), and enables quantitative analysis of complex phenomena using simple-to-use elements (teacher 1).

In several of his statements, teacher 3 emphasizes the importance of the *collaboration support* integrated with the modeling support as Cool Modes provides it. He gives two examples from his course in stochastics:

- The transition from relative frequencies to the notion of mathematical probability requires a *high amount of data*. The condensation of data, enabled through the sharing data with collaboration functions, can contribute significantly to this.

- Some experiments in the field of stochastics are very time consuming and require a lot of processing power (e.g., lotto experiments). Here, the distribution of calculation tasks that is possible with shared application instances might be valuable.

The degree of usage of the synchronization functions of Cool Modes differs between the three teachers: teacher 1 has used a basic variant (workspace wise sharing only), teacher 2 did not use the synchronized mode at all due to the speed of the computers, and teacher 3 employed even the partial (node wise) synchronization options. A common question to all the three was in how far the (practically experienced or hypothetical) degrees of flexibility in sharing models can contribute to the success of joint modeling tasks. The answers to this, as quoted in table 8.6, outline a generally positive attitude, and do also show the diversity of usage options that the teachers have in mind concerning this feature.

The last subtopic of task interoperability that the teachers addressed in their statements was *phase support*. The versions of Cool Modes that they used in their lessons did not foresee different interaction modes, so that feedback on the use of this option is not available. However, two contrary statements were made in this respect: teacher 2, being directly told about the newly available interaction modes and asked about his opinion, replied:

> *"I am just now thinking about this question for the first time. Obviously, it has never disturbed me that multiple modes were not there."*

He even pointed out that for him, the integration of model construction with simulation is an important aspect. On the other hand, the different usage modes identified by teacher 3 (cf. subsection 8.2.3) may indicate that a tailored support for these modes might be helpful for the intended educational scenario. Thus,

Table 8.6: Teacher's views on highly flexible synchronization options in Cool Modes

Teacher	Statement
1	*"I can hardly imagine this at the moment, because I have never used something like this, where you practically share results and jointly work on a model. With this option I'd require help myself to say something qualified. Generally I think it's interesting, because there is something like a conversation in the lessons where a problem is presented and discussed among the students how to deal with it and approach the solution. And that could of course be done on the level that the students do not only give their input in form of discussion contributions, but also as entries in a jointly used model."*
2	*"Under the prerequisite that the the technical things work, I'd say this can make sense. [...] Assumed that you have student groups that have no problems, then you could show to them only the sector that is more complex, and the others would have the section which is easier to work on. [...] The co-operation via computer is superior to the traditional group work, if the handwriting is completely usable, if you have the option of sending them something. And they have the chance to work on it. If this whole network works, then it makes more sense than the conventional method. If it doesn't work, the whole thing looks different."*
3	*"I have used a prototype of this partial coupling with the prototype of the result collector in stochastics. This co-operation mode has an enormous potential for collecting and joining the results of group work. [...] My experience with my seminar has shown that it can be very interesting to give access to nodes to different subgroups, in order to systematically share information about data in different layers of a system."*

the ambiguity that was identified on the theoretical level (modeless interaction vs. modeling phase support) has equivalences also in the practical feedback given by the teachers.

8.2.5 Social Interoperability

In addition to the task specific collaboration aspects discussed in the previous subsection, the teachers also gave a number of statements concerning social interoperability on a more general level. Here, the following types of statements can be distinguished:

1. Remarks about the system's flexibility concerning collaborative usage
2. Educational collaborative usage scenarios of Cool Modes

All three teachers gave statements about the possible ways of using Cool Modes as a means for collaboration. These statements were related to both their experiences and also their estimation of hypothetical usage modes which they did not yet implement. Interestingly, all the teachers explicitly mentioned a not technically supported group work mode (i.e., two or three students working together on one computer) as one option which is valuable in the classroom. Here, teacher 1 had an indeed very positive impression:

> *"They have been sitting before it, and they have communicated. Many of the things that are my task otherwise, e.g. solving thinking problems, they did among themselves then."*

Teacher 2 saw this point a bit more critical and remarked that his experience showed that only fifty percent of the small groups indeed discussed using the medium. Teacher 3 expressed positive experiences, and states that the shared monitor offered a joint focus for work. In his opinion, the created visual model is

> *"[...] an extract of the problem solution, which one can fairly easy discuss."*

As all teachers pointed out that the advantages of students working jointly on one computer are limited to groups of two or three students (due to space problems and limits concerning interaction options with the application), it is apparent that they welcomed the option of synchronizing Cool Modes instances.

Teacher 1 expressed that the students used this option intensively as a means of co-operation to *"communicate ideas and pass changes in real time"*. Yet, he states that fine granular options for synchronization (e.g., sharing only partial models) might require a level of systematic activity planning skills that some students do not have. As pointed out already in the previous subsections, teacher 2 did not use the application coupling functions due to technical problems. His estimation is that, assuming the infrastructure is appropriate, this system function can be a *"valuable medium to enrich a teaching method."* As already pointed out in the previous subsection, he furthermore believes that an inner differentiation of the class can be supported using advanced partial sharing mechanisms. Finally, teacher 3, having experience also with partial model sharing scenarios, states that the flexibility the system offers in this respect allowed him to set up advanced collaboration settings, in which small groups worked together intensively and added their results to a "pool" shared by the whole class. His reports about these use cases were generally positive: he reported on very engaged students that actively shared their work results with the rest of the group. A point of criticism is that the re-use of

other's results might have been more intensive: in his lesson design, he intended students to use the results of peers in order to plan their own activities. This was not observable frequently.

Apart from these views on the options about collaborating via COOL MODES in different forms, the teachers also gave statements about particular situations in which they consider the synchronization functions of COOL MODES as a valuable means to implement educational approaches.

Here, teacher 1 and 2 identified in particular the option of having the better and the weaker students collaborate as a good approach. Teacher 2 imagines that here, the application synchronization might be a suitable technique, allowing better students to adopt "tutor" roles and observe other's work without having to walk through the class. Teacher 1 reports on his experiences with this type of working groups as *"very fruitful co-operation sessions"*.

The statements of teacher 3 go beyond this (relatively) simple usage mode of the COOL MODES synchronization functions. His statements demonstrate that he used the functions to orchestrate complex group scenarios, manifested in a collaboration process and groups dynamically assigned using the synchronization functions: in one of his scenarios, the class was split into two groups. The first group worked in one collaboration session and internally organized its division of labor according to the task. The second group was divided into several subgroups with different task assignments, and included explicit "integrator" and "presenter" students, who were member of both subgroups and could participate in the corresponding collaboration sessions. Teacher 3 states that *"all this worked fine to a large extent."* (cf. Kuhn, Jansen, Harrer, and Hoppe (2005) for a more detailed description of the lessons). This is a very positive indication, taking into account the argument (brought up by teacher 1) that students of the corresponding age are typically not masters of systematic work.

8.2.6 Pedagogic Assessment

As a last point of the interview analysis, this subsection contains general statements given by the teachers concerning the usefulness of COOL MODES in school lessons. These statements can be classified into the following four areas:

1. Practical changes in the lessons
2. The role of the teacher in the lessons
3. Aspects related to the student's learning
4. General statements

Concerning the first point, the three teachers share the opinion that the use of the COOL MODES software in their lessons made new options available that differ qualitatively from "conventional" approaches. Giving arguments for this evaluation, both teacher 1 and teacher 2 emphasize the dynamics of the models that students can build as an essential factor. Both believe that the ease of designing, modifying, and simulating models is an important difference to ordinary (i.e., non-computerized) methods. Teacher 3 also shares this viewpoint, and adds that indeed some usage scenarios were not possible before: the execution of a very high number of experiments in stochastics is *"very difficult on paper, and in the border areas not possible."* According to him, the system did indeed bring new options. In addition, he mentions a number of particular areas where the COOL MODES system significantly facilitated certain processes. These examples include the collection of experiment data created by the students working in distributed subgroups - here,

Table 8.7: Teacher's views on their role in lessons conducted using Cool Modes

Teacher	Statement
1	*"After knowing myself how this all works, the lessons indeed had a different quality. I could observe group and learning processes"*
2	*"You take back yourself more and hope that he students work on that independently."*
3	*"First of all, my teacher role changed because of the distributed usage. Some students worked independently at the computer, so that I could work more directly with the other subgroup. [...] It is similar to other forms of group work. You have an intensive exchange with some groups, yet none at all with others."*

the collaboration support functions were an essential factor that helped replacing a lot of tedious paper based work.

Similar to the first aspect, the teachers also give very similar estimations about their role in the lessons when the Cool Modes software was used. Though there are certain nuances in their statements (cf. table 8.7), they all report on a change in the teacher role that goes along with a more active student role. A summary of their statements is that they all see Cool Modes as a system which enables small student groups to work together, and thereby relieves the teacher from certain tasks. This is fully in line with the objectives of the SEED project, where the activities of the teachers have their origin in (cf. subsection 8.1.1).

Apart from these changes in their role, all three teachers also gave statements about their perception of the student's behavior in the lessons, and also interpreted this with respect to learning success. Here, teacher 1 clearly states that the new options coming with the use of the Cool Modes software (i.e., quantitative modeling and experiencing large impacts of small model changes instead of qualitative extrapolations) did contribute to the learning success of the students. In particular, he mentions the *discussion* of results that the software facilitates as an important aspect. Teacher 2 points out that he believes (yet, without having a "control group") that the use of the software had a positive impact on the learning success. His estimation is that this is largely due to the option of *dynamically simulating* models, which he considers a very motivating factor. Teacher 3 shares this opinion. Comparing the collaborative modeling activities using Cool Modes with other forms of teaching, he summarizes:

> *"The students get a shared focus faster. [...] The process is experienced live, not only in the head."*

In addition, he states that he experienced the students indeed *trusting* the computer (in his role of generating random numbers). Based on research literature, he initially had the fear that the students would not accept the computer as a replacement for "real" experiments - yet, based on his experience working with Cool Modes in his classes, he could not confirm this problem. In the contrary, he believes that the use of the system was indeed helpful for the students, as the model execution (with a high number of repetitions) confirmed their hypotheses built on a theoretical level.

Finally, all teachers gave some general statements about their conception of Cool Modes as a tool for classroom use. The previous subsections already outlined

the positive attitude all three teachers have. In particular teacher 1 and 3 report very positively and will continue using the system - teacher 1 even puts this as

> *"You really cannot go back beyond this now. That would be a step into stone age."*

Teacher 2 gives modestly positive statements. The points of criticism he gives are related to the use of computer tools in the classroom in general instead of Cool Modes as a specific application:

- The speed of the computers did not allow for using all the functions of the system. In particular, the collaboration support could not be used.
- The available internet access in the classroom had disturbing effects on the lesson, as some students were distracted.
- A really fruitful use of the system requires equipment (a projector and tablets for handwritten input) which is usually not available in all computer rooms. The ideal equipment, an electronic whiteboard, is expensive.

Compared to this, the points of criticism that teacher 3 pinpoints are very concrete and related to Cool Modes as a specific application. He demands:

- Better process documentation options: the pure workspace oriented approach is well suited for collaboration, but has disadvantages when it comes to presenting solutions that consist of iterative model refinements.
- Integrated help functions which explain to the students the function of single modeling language primitives. There should be some support for such functions in the framework system, although the concrete help texts are of course specific to Plug-Ins.
- A connection of Cool Modes to structured archive systems (i.e., asynchronous work support mechanisms) would augment the usability of the system in the classrooms.

The third point is under current development and is shortly discussed in the final chapter of this thesis, and the first point is part of a separate PhD project.

8.3 Programmer's Views

As mentioned in section 8.1, there are currently more than 40 Plug-Ins for the Cool Modes system. The far majority of these has been developed in the context of the COLLIDE research group at the University of Duisburg-Essen. A number of programmers have been involved in these developments.

The personal opinions and conceptions of the Cool Modes framework system that these programmers have based on their work with the system is a valuable source of information that can be used to evaluate the fulfilment of some "soft goals" of this thesis, in particular those that are related to questions of expressiveness, reusability options, flexibility and (programming level) usability of the framework.

For this reason, I have conducted interviews with five programmers that were intensively involved in the development of Plug-Ins. Similar to the teacher interviews, the programmer interviews were conducted as focused interviews (cf. section 8.2). Table 8.8 shows the interview guide, and table 8.9 contains the initial coding categories.

Table 8.8: Guide for the programmer interviews

Topic	**Subtopics**
Applications and Programming experience	Programming experience (Java / general) Estimation of skills and weaknesses in programming Developed Plug-Ins Were these Plug-In rich in semantics?
Expressiveness and Interoperability	Were any "workarounds" necessary? Did the system support / restrict the development? Do Plug-Ins work together with others? If yes: How realized? Did problems occur?
Reusability	Which system classes or libraries were used? Areas of greatest support? Difficulties in implementing load or save functions? Was the cooperation support in the framework used?
Assessment	Problems integrating the Plug-Ins into the system? Estimation of requirements for developing Plug-Ins Programming without the framework: consequences? General suggestions for improvement

Table 8.9: Coding categories for the programmer interviews

Code	**Description**
Context	Statements about own experience and skills, and the developed Plug-Ins.
Expressiveness	Statements that relate to things that the framework does allow or not allow, and statements about functionality provided by the framework.
Interoperability	Statements related to the re-use of system classes, and re-use of functionality across Plug-Ins.
Ease of Use	Statements that refer to the difficulty of programming with the Reference Frame framework and Cool Modes, including general suggestions for improvement.

Table 8.10: Backgrounds of the interviewed programmers

P.	Experience (Java)	Skill Level	Weaknesses	Strengths
1	5 years (1 year)	medium	user interfaces, complex algorithms	object oriented design
2	10 years (7 years)	pretty good	user interfaces	transactions, distr. systems
3	11 years (3 years)	quite advanced	distributed systems	-
4	15 years (7 years)	very good	user interfaces	networking, distr. systems
5	10 years (4 years)	quite advanced	networking	user interfaces, XML

8.3.1 Background and Context

The five interviewed programmers were selected for interviews because they are the persons with most experience in programming Plug-Ins for Cool Modes. They all volunteered for the interviews and are (at the time of this writing) associated to the COLLIDE research group: programmers 1, 2, and 5 are master students and work as student assistants, and programmers 3 and 4 are PhD students working as research assistants.

The background of these five programmers is listed in table 8.10. All information, in particular also the strengths and weaknesses and the level of experience, are according to the self-classifications given by the programmers.

The table shows that the interviewed programmers are generally already on an experienced level. This practically eliminates impacts from Java language characteristics (i.e., issues not specifically related to the system to be evaluated) on their statements about the Reference Frame framework, as they are familiar with the general capacities and limits of the programming language.

The table also reveals that the areas of expertise vary considerably among the interview partners: some declare their strength to be in the fields of distributed systems and networking, while others nominate these areas as their weaknesses. The field of user interface programming plays an analogous role.

Finally, table 8.11 lists all the Plug-Ins that the interviewed programmers developed, co-developed or revised. This table shows that (except for programmer 1, who is an important interview partner due to its only medium level of programming skills and experience, compared to the others) all interview partners have significant experience in programming Plug-ins for the Cool Modes system, which substantiates their judgements and assessments. As already stated at the end of subsection 7.1.1, nearly all the implementations have been based on the program centered approach (sometimes, an inclusion of external data files has been used). Therefore, the interviews can only give results on this approach for Reference Frame specification.

Most of the Plug-ins contained in table 8.11 are mentioned as examples at various locations within this thesis - the remaining ones are not further described and included in the table for reasons of completeness.

8.3.2 Expressiveness and Reusability

The first analysis category for the interviews is related to the expressiveness of the Reference Frame approach and its implementation underlying the Cool Modes system. Here, statements belonging to three different subcategories were made:

Table 8.11: Plug-Ins developed by the interviewed programmers

Programmer	**Plug-Ins**	**Degree of Semantics**
1	Learning Phases, Educational Modeling	very low
2	Functions, External Devices, CampusCouples, SMS, Tutor	varying, from very low to very high
3	System Dynamics, Petri Nets, Discussion, Analysis, QOC	highest degree of all that currently exist
4	Biosphere, Moon craters, E-Mail listener, Board games, Discussion, Language Mixer	varying, from very low to very high
5	Maze, System Dynamics, Functions, Stochastics, Discussion, Tutor, Simple UML	varying, from very low to very high

1. Statements about the expressiveness of the Reference Frame concept as such: options and limits of the modeling languages that can be implemented as Plug-Ins within the architecture.

2. Available and lacking functions of the overall system implementation, including the management of Reference Frames and other services provided (or desired) in the software backbone.

3. Use of the provided abstract base classes.

The range of statements given by the programmers differs considerably between the first two subcategories. Concerning the first one, the expressiveness of the Reference Frame concept, only little criticism was made. None of the five programmers felt that the Reference Frame encapsulation embedded in the Cool Modes system restricted him in any major way during the implementations of visual modeling languages. Programmer 1 states that he *"did not feel particularly restricted"*, programmer 2 also considers the framework *"rather a help than a corset."* Programmer 3 emphasizes the *"very high usability"* of the system, programmer 4 states that *"there were no restrictions"*, and programmer 5 believes that *"the system does offer a lot of options to set up graph based modelings."* Only two minor disadvantages were brought up:

- Programmer 4 referred to an older version of the Reference Frame implementation which did not provide the option of defining Plug-Ins without Palette, and

- programmer 5 said that the general notion of Reference Frames should also include interfaces to system-wide context sensitive help functions

None of the programmers reported on a case where the Reference Frame approach was insufficient for developing a graph based modeling language. This is in line with the design choices discussed in chapter 4 and 6: here, the flexibility of the implementation with respect to supported modeling languages has been a primary factor.

Concerning the second subcriterion, the functions and expressiveness of the application framework Cool Modes rather than the options of designing Plug-Ins, the answers given by the programmers are quite heterogeneous: all programmers mention both a number of positive aspects (i.e., important functionality that the system provides), and also points of criticism in the form of functionality that should be added or modified.

The mentioned positive supportive functions of the system architecture and framework as conceived by the programmers are listed in table 8.12. These are essentially based on answers to direct questions about important system services that have used, and also on indirect questions (e.g., "Imagine the framework system did not exist, and you would have to implement a tool with same functionality as you did using the Plug-Ins - where would you expect difficulties?").

Obviously, the table is not complete in the sense that the programmers have indeed used a number of functions that they did not mention (e.g., all of them have explicitly used the event mechanism and implicitly the Plug-In management, but only one mentions it). It can be assumed that a more systematic way of asking (e.g., with a fixed list of functions and direct questions about their importance) is likely to lead to different results - yet, the table points out the system functions that the programmers were aware of and considered important.

Though the single points mentioned by the programmes depend on their estimation of functional importance and also their usage context, the table as a whole

Table 8.12: System functionality considered important by the programmers

	1	**2**	**3**	**4**	**5**
Cooperation support	x	x		x	x
Drag & drop functions	x		x		x
File handling (XML) support	x		x		x
Conceptual graph structure	x	x	x	x	
Graph view and layout	x	x		x	
Node duplication	x				
Plug-In management	x				
Localization		x			
Syntax rules		x	x		x
Usage modes		x			x
Workspaces			x		x
Palette concept			x	x	
Event mechanism and semantics			x		

reflects well the variety of features that the system offers to the Plug-In developers. Cooperation support and graph structures are mentioned as important functions by the majority of programmers - for both areas, the framework provides a layer to easily access the functionality of the underlying libraries. In addition, all the other core aspects discussed in the implementation chapters of this thesis (syntax and semantics, visual attributes and layout, Palettes, etc.) have been mentioned. Some of these concepts will be revisited in the interoperability and reusability discussion in the next subsection.

In addition to these positive statements about available and important functionality provided by the framework, the programmers also mentioned a number of drawbacks and desired functionality extensions:

1. Both programmer 1 and programmer 5 expressed the wish for a better support of hierarchical structures (nodes containing graphs). This first point of criticism is related to the expressiveness of the underlying graph structure and does not directly criticize the Reference Frame framework as such - with a changed underlying graph library, such an extension would easily be possible.

2. Programmers 1 and 5 stated that there is a lack of communication options between Reference Frames and the content of workspaces: nodes in workspaces cannot easily access "their" Reference Frame, e.g., to modify it dynamically. This is indeed not foreseen in the approach, as the Reference Frame and its Palette are defined independently of concrete models, and in addition there is no direct relation between node or edge types and Reference Frames on purpose. However, both programmers gave quite "extreme" examples of Plug-Ins (Cool Modes being used a frame for a computer game and as a visualization of dynamic Palette content details) that would benefit from these communication options. They stated that a "regular" usage of the framework does not require these channels.

3. Semantic interoperability was addressed by programmer 3, who poses the question whether more controlled methods for data exchange between different node types contribute to interoperability between Reference Frames. This will be discussed in detail in the next subsection.

4. Programmer 4 stated that hot deployment (i.e., dynamic adding of Plug-Ins at runtime and loading new versions of a Plug-In as they become available)

would be desirable. Although the Plug-In mechanism as such is dynamic in the sense that the user can add Plug-ins at runtime, the criticism is legitimate in the sense that the employed mechanism does currently not consider newly available class resources in the search: only the XML based Plug-In definitions are treated this way. A modified class loader could solve this problem of dynamic classpath modification.

5. Programmer 5 requests a more flexible drag & drop mechanism that also works with hierarchical structures (cf. first point). This is indeed a possible degree of freedom that could be embedded into the framework - however, this would have to be done taking into account that some programmers explicitly appreciate the "simply working" drag&drop in its current non-parameterizable form (cf. table 8.12).

6. An improved library for sending e-mails from the system (e.g., containing the XML file and a snapshot of the workspace) and invoking other external tools (e.g., browsers) were demanded by programmer 5. This point of criticism is not related to the core functionality of the framework.

Apart from these particular aspects, all five programmers report on a generally high level of support that the framework has provided them with. A specific positive aspect was expressed indirectly by programmer 3: he believes that both the underlying graph and synchronization libraries should be replaced by more standardized ones. Being asked whether he thinks that this would require much refactoring work in the existing code, he replied:

> *"No. They all base on classes that would have to be changed then. But one would not have to change the classes themselves. From this, I think that it would not be so much effort."*

This statement emphasizes, beyond the role as direct provider of functionality, the role of the framework as a bridge between the modeling language specifications and the underlying libraries: even if the latter, containing essentially used functionality, are exchanged, this does not affect the Reference Frame definitions as such.

The third subcategory is related to the use of the abstract base classes provided in the framework. Of course, there is no really sharp borderline to the second subcategory, as a reuse of functionality (provided somewhere in the framework) may sometimes go along with a reuse of classes as base classes.

One typical question in the interviews was which system classes or libraries the programmers were in touch with and/or consider important. Here, the answer was homogeneous: all five programmers stated that they preferred working with the provided abstract base classes for Reference Frames and Palettes. None of them used the XML based specification option, and neither did any of them directly implement the Java interfaces (cf. subsection 7.1.1). This expresses that the functionality provided by the base classes was widely accepted. Being asked directly about the reason for their choice, all programmers replied that the implementations available in the base classes drastically reduce the work for developing a Plug-In. Programmer 3 even stated that he never really looked at the code of the framework system: extending the available base classes was always sufficient for his developments.

This degree of reuse of the abstract base classes for Reference Frames and Palettes allows the conclusion that in general, the provided functionality was welcomed by the programmers (cf. table 8.12). Detailed points of criticism were:

- Programmer 1 and 2 suggest an enrichment of the base classes with even more functionality, as most Plug-Ins share very similar behavior, which currently leads to a copy & paste usage.

- Programmer 3 addresses the problem of semantic interoperability, and discusses an extension of the core interfaces (cf. next subsection).

- Programmer 4 suggests dynamic code generation for the XML serialization, using serializer and deserializer components.

8.3.3 Interoperability

The previous subsection already contained a discussion of the software components that the programmers referred to as important and helpful, or lacking and to be improved. Apart from these general aspects, the topic of how the programmers dealt with interoperability between different Plug-Ins is important: in contrast to the language primitives specification, the syntax rules and the collaboration support mechanisms, the semantics of languages and the algorithms for implementing semantic mappings are only supported on a basic level (cf. subsection 6.4.2), and the interoperability options between different Reference Frames are kept very flexible (cf. subsection 6.3.2). As argued in the previous chapters, these design decisions were made in order to not restrict the development of Reference Frames. Yet, due to the absence of a uniform "interoperability algorithm" (which is available in comparable systems, cf. section 3.2), it is interesting to see how the programmers did indeed make use of the flexibility - or whether they would have preferred a more controlled and guided architecture.

The statements given in the interviews are related to the following two subtopics:

1. Approaches and experiences concerning interoperability between different Reference Frames.

2. Usage of the available abstract base classes and their embedded interoperability support.

All five programmers gave statements related to the first subtopic, which can be clustered as follows:

- Programmer 1 did not use any advanced model semantics (cf. table 8.11). Yet, one of the Plug-Ins he developed *extends* an already existing one, both on the node/edge level and also on the Reference Frame level. Thus, his implementations constitute an "is-a" relation between the two Reference Frames.

- Programmers 2 and 5 report on semantic interoperability reached on a technical level through a dedicated *interface* used to connect Plug-Ins.

- Programmers 3 and 4 state that they looked at the data flow within heterogeneous models also from a general perspective, and found out three different approaches (via *interfaces*, the *Java reflection* mechanism, and MatchMaker).

Although these answers have to be interpreted carefully, since programmers 3 and 4 have worked on the problem of semantic interoperability also on a conceptual level to some extent, whereas the other programmers took more straightforward approaches, the variety of answers, together with the heterogeneity of developed Plug-Ins, underlines the flexibility that the Reference Frame approach offers with respect to semantic interoperability.

The programmers that belong to the first two clusters (i.e., use of extension between Reference Frames, and interfaces to interconnect languages) do not report on serious problems. Programmer 1 just states that *"sometimes you have to be careful about which palettes feel responsible for which nodes"*: this mirrors the design

decision of having loose associations between Reference Frames and node or edge types.

Both programmer 3 and 4 mentioned that they worked together on a general mechanism to allow for data exchange in heterogeneous models. This work has lead to the development of a special Plug-In, providing a node designed for data exchange: using a general mechanism, this node imports data from other nodes and is capable of exporting it, provided that the target nodes follow certain conventions. Technically, the programmers report on different underlying mechanisms (cf. bullet list above).

The fact that these developments were indeed possible within the framework demonstrates the flexibility of the Plug-In concept: it was possible to define a "Meta-Plug-In" to interconnect models without having to modify the system core. Apart from this, the opinions of the two programmers vary. Programmer 4 gives positive statements:

> *"We always let the data flow along the edges. [...] All three options worked quite well. They had advantages and disadvantages, but these were inherent, not due to the framework system."*

Programmer 3 adopts a more critical perspective. According to him, interoperability in heterogeneous models is *"not so easy"*. He considers syntactic issues fully covered, but semantics and model execution only supported as long as this is possible based on the event mechanism:

> *"The syntax part is definitely there. You have to do the semantics yourself. As long as that works with events and notifications, it is also available semantically, but if you want data exchange you have to do something yourself."*

In contrast to programmer 4, he raises the question whether the integration of a generic data exchange (or: semantics representation) format would enhance the interoperability options in the system. Yet, (although having done some work in this field before the interview) he does not have a clear solution:

> *"Concerning interoperability, I have thought of whether general input and output methods should be available, that every node writes its values as primitive types and is also able to accept them. [...] One should investigate if this increases interoperability or if this is nonsense. I think one could still do something about the data exchange between nodes."*

This statement reflects some of the decisions made in system architecture design: there is no uniform and exclusive way of data exchange or semantics representation, and neither are general control algorithms (apart from the event propagation) foreseen or enforced. These decisions contribute to the flexibility of the system and the expressiveness of the Plug-Ins - however, more "guided" structures could be tailored for, e.g., interoperability between specific modeling languages. Here, some of the more theoretically oriented approaches presented in section 2.3 can have guidance character for implementations. In particular, the grammar based transformation approaches (Lara & Vangheluwe, 2004), the transformation types identified by McBrien and Poulovassilis (1999), or the strategies discussed by Dolk and Kottemann (1993) may be worth considering. The existing Plug-In based implementation for model interoperability done by programmers 3 and 4 shows that the framework indeed supports this meta-level functionality.

Syntactic issues, in the contrary, are not discussed in such a controversial manner: with respect to interoperability, the programmers pointed out that the approach easily allows for defining "closed" and "open" languages (i.e., allowing language mixes or not) by simply using or not using corresponding rules in the Reference Frame.

8.3.4 Ease of Use

The flexibility and expressiveness of the framework as conceived by the programmers who used it has been discussed in the previous subsections. Apart from this, also the level of difficulty in using the system is an interesting factor: did the programmers feel comfortable with the libraries, or did they have major problems caused by the framework during their developments?

In the interviews, several statements related to this area have been made. Four subtopics emerged here:

1. The use of particular aspects of the system functions (e.g., cooperation support, or XML serialization).
2. Remarks about the integration of the developed Plug-Ins into the framework.
3. Specific suggestions for improvement.
4. Estimations about requirements for developing a Plug-In for Cool Modes.

These four subtopics highlight different areas of programming level usability: the first one gives answers to the level of support that the framework system gives for certain required tasks, the second one is essential because the quality of an extensible framework system is to a large extent determined by the ease of integrating new parts. The third subtopic reflects whether central services or features are missing, and finally the fourth subtopic is an additional measurement for the complexity of the library, especially when taking into account the personal experience profiles of the programmers as listed in table 8.8.

During the interviews, all programmers gave a number of statements related to the parts of the system that they considered important and have used extensively (cf. subsection 8.3.2). This fact shows that they have a positive conception of the libraries: due to the flexibility of the system, proprietary solutions would have been possible alternatively for most of the tasks. In addition to this indirect rating of the framework system quality, the programmers also gave some direct statements. These were related to two exemplary parts of the system functions: the XML serialization and the cooperation support. Both functions are quite complex on the general level: the serialization of heterogeneous models, including the parser handling, requires some experience and a number of algorithmic techniques, e.g., for loading potentially unknown elements - which is inevitable using a dynamic Plug-In mechanism. Cooperation support (i.e., synchronization), is even more difficult. Here, major parts of the work (connection, session, and user management as well as construction of major parts of the synchronization tree and initialization of basic listener instances) are done by the framework: the Plug-Ins only have to specify the node and edge dependent parts. Thus, positive statements of the programmers about the support level could be expected.

This expectation was met. Table 8.13 summarizes the statements that the programmers gave concerning the level of difficulty. The table shows that the positive statements clearly outweigh the few mentioned disadvantages.

As argued before, the integration of Plug-Ins into the framework is a critical success factor: the dynamic inclusion of system extensions should be as easy as

Table 8.13: Programmer's statements about serialization and cooperation support functions

P.	Serialization	Cooperation
1	XML serialization works smoothly, library decreases work.	Better documentation of MatchMaker details needed.
2	Rather helpful and transparent library, but could be lighter. Inheritance might bring problems.	Intuitive interface, but difficult to debug. Problems with event cycles (local and remote events)
3	Very easy. The basic features are all there.	Basic functionality is there for free. For more advanced things, a better documentation would help.
4	Knowing DOM, this is very easy.	Clear structure and interfaces, works fine when used consistently. Debugging is problematic.
5	Problems at the beginning, but got into it quickly.	Very easy, also for non-experienced programmers.

possible for the developers of the extensions, in order to not require them to know details about the internals of the embracing framework.

Therefore, one topic in all the interviews was whether the programmers had any problems integrating their work into the framework. Here, the answers were very positive and substantiated the quality of the dynamic Plug-In mechanism and the management of multiple Plug-Ins by the framework (cf. subsection 7.1.2): Programmers 2, 3, and 4 did not report on any encountered problems . Programmers 1 and 5 mentioned only minor issues (errors caused by wrong specifications of resource locations, requests for better debugging options), but also expressed that they had no general problems.

The third subtopic discussed in this subsection is which specific propositions for extensions (in order to simplify the use of the system) were given by the programmers. Here, the following three points were addressed:

- Programmer 2 demanded a "Reference Frame Builder" to simplify the development. He can imagine a visual environment to specify the basics of a Reference Frame and its Palette, generating a code template. Programmer 1 and 4, however, believe that such an option is not useful.

- Programmer 2 suggests a more elaborated document model: currently, all workspaces and loaded Reference Frames make up the document. He proposes an extended model, allowing a Cool Modes instance to contain multiple documents, each associated to a set of Reference Frames and workspaces.

- Programmer 4 and 5 addressed lacking archive functions not only on the document level, but also for Plug-Ins and for other resources: according to them, central places to store and retrieve system extensions could facilitate interoperability and reusability of the framework.

These topics are all relevant and should be considered for further system developments (cf. next chapter). Yet, evaluating the quality of the Reference Frame approach and the architectural framework proposed within this thesis, an important factor is also what is *not* mentioned: none of the programmers spoke about

Table 8.14: Programmer's estimations of requirements for developing a Plug-In

	1	**2**	**3**	**4**	**5**
General Java skills	∘	∘	∘	+	∘
Concepts of Object Orientation	+	+	+	+	+
Distributed Systems techniques	∘			+	
XML handling	∘			∘	
Design Pattern: MVC	+	+		+	+
Design Pattern: Observer			+		+
User Interface programming			∘		∘
Graph concept				+	

improvements concerning the specification of visual languages as such, the collaboration support interfaces were not mentioned, and neither were the model integration or interoperability functions criticized from a usability point of view (yet, this has to be seen in the light that the amount of complex and interoperable Reference Frames is not very large). This lack of criticism about the core parts, seen in the light of a generally open attitude of the programmers in the interviews, strengthens the hypothesis that the framework is indeed not too complicated to use.

This is supported by the estimations that the programmers gave concerning the requirements for successfully developing a Plug-In for Cool Modes. These are listed in table 8.14, with + standing for "good knowledge of the concept", and ∘ standing for "basic knowledge". Similar to the other comparable aspects in the interviews, the questions were asked in an open manner, i.e. no specific categories were suggested.

An analysis of the table yields two things: first, the concepts mentioned most frequently are on a general level (programming skills, object orientation, design patterns). Specific skills, which would definitely be needed for building a collaborative modeling tool "from scratch" (e.g., knowledge about programming distributed systems, or user interface design) were rarely mentioned, or even not mentioned at all (e.g., language integration issues). In combination with table 8.12, which lists the important elements of the framework as conceived by the programmers, this result emphasizes the impression that the framework serves the purpose of relieving the programmers from "lower level" tasks by certain architectural constructs (which, in turn, explains the object orientation and design patterns as required knowledge), and allows them to concentrate on the task of specifying the modeling language as such, which can then be used in the application framework in an integrated manner.

A particularly surprising detail result comparing table 8.14 with table 8.10, which contains the skills of the programmers as they identified them (usually at the very beginning of the interviews), is that indeed *none* of the five programmers stated that his personal weak fields (typically user interface programming or distributed systems technologies) were an important requirement essential for developing a Plug-In for Cool Modes. There are several alternative explanations for this effect: e.g., they might simply not want to admit that their weaknesses were in important areas. Yet, considering the answers given to the used system functionality (table 8.12) and the finally successful outcome of the programming work (cf. section 8.1), a clear guidance and helping function of the framework can be noticed.

8.4 Summary

This chapter evaluated the Reference Frame approach and its proposed architecture and system implementation with respect to the "soft criteria", which, being related to the concrete usage of the system, could not be proved or shown by logical arguments.

The intended *flexibility* and *expressiveness* was addressed both in section 8.1, presenting some usage scenarios of the Cool Modes system, and also in the statements that the programmers gave during the interview. The results in terms of both variety of usage scenarios and programmers feedback are fully in line with the design decisions taken (flexibility as an essential criterion), and substantiate the fulfilment of the flexibility criterion.

8.4.1 Programmer's Interviews

In general, the statements of the programmers were not uniform. This is not surprising, as they have different backgrounds and experience, and have implemented very differently targeted Plug-Ins prior to the interviews. A number of propositions for potential improvements have been made, related to both functional aspects of the framework, and structural changes to improve usability. The core topic of interoperability was seen critically by one interviewed programmer, who believed that a more guided approach (versus the delegation of algorithmic control and parts of the interface specification to the Plug-Ins) might be beneficial. Indeed, this alternative approach, which is taken by comparable systems (cf. discussion in chapter 3) might contribute to a more controlled and centralized management of data flow in heterogeneous models, but it risks reducing expressiveness and flexibility.

Despite these negative points, the general outcome of the interviews with the programmers conveys a positive image: the Reference Frame concept as such was considered useful and well suited for the implementations, and a convenient use of the framework was attested. The programmers appreciated the functionality provided by the system (in particular through the abstract base classes). According to their statements, the structural design of the system does indeed both facilitate the development of single Plug-Ins in many respects, and also easily allow for an integration of the Plug-In with others. These interoperability and wiring options, together with the dynamic Plug-In mechanism, meet the "soft" programming level criteria listed by Roschelle et al. (1999) (page 20).

8.4.2 Teacher's Interviews

Apart from the experiences of the programmers, which serve as an indication for the quality of the approach and the system implementation, the second source of information used in this chapter were the teachers who have used the Cool Modes software in their regular lessons, with varying topics (from biology, mathematics, and computer science), Plug-Ins, and classes.

A formal evaluation of these usages was not conducted. Such an approach would have helped clarifying concrete detail questions. Yet, the largely uncontrolled character of the lessons (in the sense of a high number of uncontrolled variables) and the lack of data, in particular control groups, prevented such an approach - in addition, the inevitable dependency on the concrete Plug-Ins used would have been a problem.

Instead, the feedback of the teachers as given in the interviews has been used as a source of information, which turned out to be very valuable. Uniformly, they confirmed the theoretical positions discussed in the introduction to this thesis (usage of modeling and collaborative modeling in education) from their practitioner's point

of view. They also reported on a sufficiently high usability of the system (i.e., the system can be used successfully in classrooms), going in line with a high degree of flexibility concerning its usage.

The general attitude the teachers convey differs: two of the teachers adopt extremely positive positions and criticize only details of the software system, whereas one teacher reports on mixed experiences: the computers in his school were too old to allow for a reasonable work, and in addition he was also lacking other equipment he considers essential for a really successful use of the system (e.g., handwriting tablets and an electronic whiteboard). Apart from these quite technical, but practically important aspects, this teacher shares the positive attitude of the others.

In particular, all three teachers state that they see task interoperability and social interoperability supported through the Cool Modes system: they report on differently designed collaborative settings, from very simple forms (with students sharing a computer) to advanced ones which make use of partial model sharing mechanisms, and confirm the appropriateness of the available synchronization mechanism as a form to foster collaboration via shared visual languages (or models) in the classroom use. In addition, they report on usages of various task interoperability functions of Cool Modes. In particular, the dynamic simulation options and the mixture of elements from different languages were positively mentioned.

Despite aspects of criticism (e.g., a lack of asynchronous storage mechanisms, help mechanisms, and functional limitations of the Plug-Ins used), these positive statements together with the number of system usages in classrooms and the continuing interest of the teachers indicate that the Reference Frame approach and its implementation in form of the Cool Modes system can indeed contribute to social and task interoperability in the targeted educational scenarios.

8.5 Conclusions

Supplementing the descriptive interview summary as contained in the previous parts of this section, this final subsection of the evaluation chapter presents some assessments and conclusions that can be made based on the evaluation results. In order to keep this list short and readable, these are not fully justified here (see the corresponding sections in this chapter for details).

- The Reference Frame approach and its Cool Modes implementation have proven to be highly flexible and expressive, both through the variety of existing usage scenarios and also as an outcome of the programmers interviews.

- The openness of the system, achieved by means of the easy-to-use Plug-In approach, is an important success factor. Although the external interfaces of Cool Modes (i.e, the data exchange with other applications) can still be improved, the existing level of interoperability already allows for using the tool quite flexibly.

- Both from the programmers and the teachers point of view, the system is characterized through a high usability. Yet, improvements are still possible here (e.g., integrated help functions, or additional services in the system architecture).

- According to the teachers assessment of the lessons they conducted using Cool Modes, the system can indeed contribute to supporting learning through its provision of collaborative modeling functionality. A typical effect of using the tool in the classroom was a teacher role change towards a more observing attitude, leaving more time for individual care for the needs of single students.

- The collaboration support functions in the system architecture are easy to use and facilitate the task of developing a modeling language for networked use. The flexibility of the collaboration options (e.g., using phases and synchronization contexts) have been assessed positive by the teachers. However, there judgements were only partially based on their own classroom experiences (some had more the character of opinions). More detailed and/or substantiated results will require further intensive usage of the advanced system options.
- The theoretically proven options for syntactic and semantic interoperability between modeling languages have been used by the programmers in a variety of ways. The opinions about the "loose coupling" paradigm in the system architecture differed considerably. Therefore, the question if a more guided and controlled implementation of the Reference Frame concept can facilitate interoperability even more can not be finally answered.

Apart from these (largely positive) conclusions that can be drawn based on the interviews, further issues which rather have an outlook character than being directly related to the core parts of this thesis are discussed in the next chapter.

Chapter 9

Summary and Discussion

The Reference Frame approach and implementation to support collaborative modeling with graph based representations is the main contribution of this thesis. Motivated from potential applications in education, the work is methodologically rooted in computer science, at the intersection of fields like metamodeling, software design, distributed systems, and visual languages in HCI contexts. In this combination of computer science methods applied with a view towards educational scenarios, the work is in the tradition of research fields like groupware, AIED, and CSCL.

This final chapter of the thesis is organized as follows: section 9.1 summarizes the major line of argumentation and the main contributions of this work, and section 9.2 discusses certain important aspects in the light of possible alternative solutions and further research opportunities.

9.1 Summary

The motivation for the work within this thesis is sought from the field of education and educational technology: different lines of argumentation suggest that computer support for collaboration using graph based dynamic representations ("models") might significantly support learning.

Based on this background and initial motivation, a list of criteria for a suitable and flexible computational approach and system implementation has been derived. The criteria can be split into two categories, both dealing essentially with different notions of interoperability:

Requirements concerning the conceptual and algorithmic approach.

Independent of the initial educationally oriented motivation, these can be summarized as *flexibility* concerning the supported graph based modeling languages, *syntactic and semantic interoperability* among the graph based representations (in particular, allowing for mixed heterogeneous structures), and a robust and easily extensible *framework* which is able to dynamically handle multiple modeling languages, generically supports work *phases*, and considers the specific requirements of *partially shared* models.

Requirements concerning the educational usage.

Most parts of the approach and system development are independent of particular usage scenarios, and so is the major part of work described in this thesis. Yet, with the initial motivation coming from the field of education, also some "softer" usage oriented criteria must be considered in order to stay conform with the origins.

Accepting the premise of not aiming at the development of a computer based learning environment as such (i.e., a system whose usage is claimed to lead to learning largely irrespective of the usage context) but at a tool that can be easily and flexibly used to orchestrate educational scenarios, these criteria can be noted as support for *social and task interoperability*: though its various degrees of flexibility (e.g. multiple representations, usage modes, partial sharing mechanisms), the system should enable learners to stay in their task and social context while interacting with each other and with the shared, dynamic artefact. Of course, also a high *usability* is a requirement that goes along with the intended practical usages.

A review of currently existing dynamic modeling *tools* (both educationally oriented ones and also generic ones) shows that there are no available solutions that meet both collaboration requirements and representational flexibility. In addition, there is currently no *theory* of graph based dynamic representations which takes into account both interoperability (this alone is already a wide field), visual aspects of the representation, and specific requirements of partially sharing these representation - the latter being a non-trivial issue under the requirement of retaining a shared semantics.

Building upon this problem analysis and review of state-of-art concerning theory and existing technologies, the major contributions of this thesis can be summarized as follows:

- Taking up concepts and approaches from graph theory, visual language theory, and metamodeling approaches (chapter 2), the concepts of visual typed graphs and Reference Frames represent formal notations for models and modeling languages used in collaborative contexts. Here, the basic notion of a Reference Frame is introduced in chapter 4. This concept comprises node and edge types, visual attributes, syntax and semantics of the language, and a synchronization context definition, which specifies the "semantic contexts" of elements that should be retained in partially shared models.

- Based on this elementary concept of a Reference Frame, several types of interoperability between Reference Frames (including the import of primitive types and the extension of Reference Frames) have been proposed, and different conceptual approaches for integrating multiple model interpretations (i.e., semantic mappings) have been discussed and compared (sections 4.5 and 4.6)

- Based on a detailed criteria list, a graph library (the JGraph) and a library for application synchronization (MatchMaker) have been identified as suitable technical foundations for an implementation of the Reference Frame approach (chapter 5). Using these libraries and an object oriented design approach, an abstract software architecture as one possible implementation of the Reference Frame approach has been proposed (chapter 6). The system offers rule based syntax specification options and support for the definition of synchronization contexts. Furthermore, it contains a lightweight event based mechanism for model interpretation (here, the simulation of models can be conceived as a side effect of the semantic mapping function), and some options for Reference Frame interoperability also on the implementation level, the latter corresponding to the relationships exemplified on the conceptual level.

- The central design decisions, in particular the trade-off between expressiveness and flexibility on the one hand, and central control on the other hand, have been discussed - in general, the conceptual approach allows for various implementations with different characteristics. The choices made in the proposed architecture were guided by the principles of striving for high expressiveness and freedom in the design of Reference Frames first, and therefore delegates

some central aspects (in particular the algorithmic control of the interpretation mechanism) to the single Reference Frames.

- The software architecture allows for different concrete applications to be built on top of it. One of these, the COOL MODES modeling framework, has been presented in detail in chapter 7. This system manages multiple Reference Frames (called Plug-Ins) and relies on the "shared workspace" metaphor. Through specific user interfaces for Reference Frames (called Palettes), the users are provided with an easy means to build heterogeneous models using the primitives that the languages provide. COOL MODES as a tool which offers multiple work phases and several forms of partial synchronization (relying on the synchronization contexts) has been described.

- In the COOL MODES system description context, different options for the definition of Plug-Ins have been discussed. Both code centered (Java interfaces) and document centered (XML files) forms are supported, and also mixtures of these forms are possible. The COOL MODES framework is capable of retrieving the differently located and defined Plug-Ins, and transparently offers them for dynamic and integrated use. The system is indeed unique in that it offers

  - fine granular synchronization options that allow for sharing model parts across different applications,
  - the external specification and dynamic inclusion of flexibly defined modeling languages, and
  - their integrated use, allowing the user to build heterogeneous structures.

- The fulfilment of the functional criteria that were related to formal characteristics of the approach and system implementation (e.g., syntactic interoperability allowing for heterogeneous graph structures, or expressiveness of the Reference Frame concept) has been shown or even formally proved. Some other more usage-related criteria needed an evaluation, which was conducted using focused interviews (chapter 8):

  - Programmers that developed a number of Plug-Ins confirmed the flexibility of the approach, the suitable design of the framework including appropriate reusable service libraries, and the working Plug-In mechanism. Contributions of the framework to model interoperability were discussed controversially (cf. next section).
  - Going back to the initial motivation for the system, it is interesting to see whether the expected benefits (social and task interoperability) are observed by the teachers: can the system really be used to orchestrate the lessons in the anticipated way?. Under the prerequisite of appropriate hardware technology, this was confirmed by teachers who used the system in their lessons.

Table 9.1 takes up the system review from chapter 3, and classifies COOL MODES according to the criteria used in the comparison. In accordance with the points discussed above, the table shows the strengths of the system with respect to supporting synchronous collaborative modeling tasks (last column) with rich and active computational representations (+ in the criterion "interoperability" and "operational semantics"). As was argued in the evaluation chapter of this and will be discussed in the next section, the COOL MODES framework does indeed leave room for improvement in the area of guidance for authoring Plug-Ins, which is the reason for the (+) symbol in the first column of table 9.1. The "-" assessment in the criterion

Table 9.1: Cool Modes in the comparison of graph based modeling tools

	Extensibility	**Interoperability**	**Operational Semantics**	**Collaboration Support**
Belvedere	(+)	(+)	-	sync: + async: ∘
Cardboard	+	-	∘	sync: (+) async: -
Co-Lab	-	+	∘	+
Cool Modes	**(+)**	**+**	**+**	**sync: +** **async: -**
Daidalos	+	∘	∘	sync: - async: ∘
DOME	+	(+)	+	-
GME	+	(+)	(+)	-
MetaEdit+	+	∘	(+)	sync: - async: (+)
ModelIt	∘	(+)	∘	-
ModellingSpace	(+)	(+)	∘	+
MULTIGRAPH	(+)	(+)	(+)	-
Ptolemy	(+)	+	+	-
Visio	+	∘	∘	sync: ∘ async: (+)

of asynchronous collaboration support will be discussed in the next section as well - ongoing research activities (which are beyond the subject of this thesis) deal with this issue.

9.2 Discussion

The approach presented in this thesis is new, and its implementation (both on the abstract architectural and the concrete system level) is a genuine contribution to the scientific field: it allows for flexible collaboration support via heterogeneous and partially shared dynamic visual structures. The work is finished and has a natural closure in the resulting framework system (whose functionality is either proved or confirmed in the evaluation), but at the same time raises several new issues and opens possible lines of research:

Reference Frame Specification.

In the interviews, both one teacher and some programmers have expressed the wish for a that a system to facilitate the development of Reference Frames and their Palettes. Such visual editors are not beyond critique from the programmers point of view - yet, in particular for non-programmers (and most teachers belong to this group!), an easy-to-use Plug-In-editor will be helpful. The basics for such a tool, however, are there: the XML based Reference Frame specification together with the template class parametrization mechanism is a suitable foundation upon which a "Reference Frame Builder" can act: the output format of this could in this case be the input XML format of the Cool Modes Plug-Ins. This would allow for an easy and comfortable specification of Plug-Ins at least with respect to attributes whose values can be declared in a simple declarative manner, comparable to the options available in the Cardboard system (Hoppe et al., 2000). On the research level, it would be interesting to investigate in how far a definition of Reference Frames can be done visually. This question is related to the field of visual programming languages, yet adds the degrees of complexity that behavior of partially synchronized and

heterogeneous structures with (a priori) unknown "other" components would have to be specified by (usually restricted) visual means.

Alternatives for Model Interoperability.

The Reference Frame approach presented in this thesis has been dealt with on three different layers, resulting in a conceptual framework, an abstract architecture, and a complete software system. Though these layers build upon each other, they are independent. For the two top layers, the different applications FreeStyler (Gaßner, 2003) and Cool Modes demonstrate this. Yet, the conceptual approach also allows for different architectural implementations. This enables different design decisions using the same underlying conceptual notions. In particular, the development of a more controlled or guided variant is possible, which prescribes, e.g., fixed model interpretation or language interoperability mechanisms (as was demanded by one of the interviewed programmers). Such an approach, as e.g. taken in the Ptolemy system (Hylands et al., 2003) or adopted in many theoretical approaches (Wang & Liu, 2003; Geoffrion, 1987; McBrien & Poulovassilis, 1999), potentially risks a loss of expressiveness. Alternative implementations of the Reference Frame approach could help further clarifying this "guidance and control vs. expressiveness and flexibility" dilemma. The + in the "interoperability" column of table 9.1 was put due to the flexibility of the current implementation, which allows for a whole range of interoperability approaches. However, more guidance might be beneficial, as discussed in chapter 8.

Model Checking.

An issue not addressed within this work is the question of model *correctness* and the mechanisms required to check this. Obviously, the step is not too far at first sight: the Reference Frame approach offers support for model syntax and semantics, and the implementation does already do some restricted model checking in that it ensures the fulfilment of the syntax rules. However, the case is much more complicated when it comes to semantics: with the heterogeneity of the visual typed graph structures and the dynamically usable Plug-Ins, the question of specifying "model solutions" (i.e., abstract representations which would allow for an automated task assessment) is not trivial. Considering the flexible but mostly uncontrolled distribution of the semantics attributes in the current architecture, the case is even more complicated. In addition, specification methods for "correct solutions" (in the best case, even in a visual form) are a challenging task. First steps in this direction have been proposed by Herrmann et al. (2003) - however, final answers are not yet available.

Asynchronous Collaboration Support.

The collaboration support methods discussed in this thesis were focused on synchronous cooperation support. However, in practice a lot of collaboration processes are asynchronous. Here, different approaches are required and have (partially) been demanded by both the teachers and the programmers in the interviews: model and Plug-In archives are one possible step in this direction. Here, some work has been undertaken by Pinkwart, Jansen, Oelinger, Korchounova, and Hoppe (2004): the heterogeneity of models (i.e., the Reference Frames "involved" in a model) here plays a role for the generation of metadata, which is in turn used for automated retrieval functions and archive based community support techniques. Yet, also this research is by far not finished: further asynchronous support functions operating on Reference Frames and visual typed graphs associated to documents (e.g., peer

recommender systems, or self-structuring community archives) are imaginable and worth investigating.

Shared Dynamic Representations.

The evaluation parts in this thesis focused on the framework level, and therefore did not concentrate on well-controlled studies with single Reference Frames (as, e.g., done by McLaren, Bollen, Walker, Harrer, and Sewall (2005) for the case of UML diagrams within COOL MODES), but on interviews with people experienced in using the system. As can be expected from the employed method of focused interviews (Merton & Kendall, 1946), the interview results served both the purpose of testing the initial hypotheses, but (due to their open character) also had hypothesis building character. In particular, the qualitative analysis gives raise to a number of concrete hypotheses concerning social and task interoperability that are worth further focused investigations:

- The teachers observed that the sharing of representations did not always lead to fruitful collaborative learning scenarios. This leads to the question about the "success" factors for using partially shared dynamic models in educational contexts. Some results in this field have been presented by Suthers and Hundhausen (2003) and Joolingen and Löhner (2001) - however, in particular the case of partial sharing in combination with dynamic models is currently largely unexplored.

- In the interviews, the teachers show different opinions about whether an explicit phase support in the software is really a benefit, or an integrated mode allowing the direct manipulation (construction of editing) of the model and its simulation is advantageous. This topic cannot be dealt with in the scope of this thesis: however, the question about the usefulness of explicitly foreseen tool-supported phases (vs. implicit phases manifesting in the actions of the users, but not necessarily in the interface of the tool) seems to be worth further research.

- Partial model synchronization has not been used largely by the teachers. Though this is at least partially due to the fact that some of the lessons have been conducted with older software versions which only had limited synchronization features, a general question is if partially shared models can indeed serve the purpose of allowing users to work with a shared understanding of the jointly manipulated artefact. Margaritis et al. (2003) doubt this and only allow for full model sharing in the MODELLINGSPACE application - however, one could argue that the shared formal semantics as guaranteed through synchronization contexts can lead to the desired effect. Here, further studies are needed to give answers.

Similar to this thesis itself, these discussion points range from educational aspects over collaboration support issues to concrete system design. For each of them, the Reference Frame approach presented in this thesis can serve as a starting point - either through the COOL MODES application being directly used or extended, or in form of its conceptual and architectural foundations.

List of Tables

List of Figures

References

Aigner, M. (1993). *Diskrete Mathematik (discrete mathematics).* Braunschweig, Germany: Vieweg.

Ainsworth, S. (1999). The functions of multiple representations. *Computers and Education*, *33*(2-3), 131-152.

Alder, G. (2003). *Design and implementation of the JGraph swing component.* Retrieved April 12, 2005, at `http://www.jgraph.com/doc/paper/`.

Aronson, E. (1978). *The jigsaw classroom.* Beverly Hills, CA (USA): Sage.

Avouris, N. (2004). *Modellingspace software final version (ModellingSpace project deliverable).* Retrieved april 12, 2005, at `http://www.modellingspace.net/Documents/PublicDeliverables/D08_final_software.zip`.

Avouris, N., Margaritis, M., Komis, V., Saez, A., & Melendez, R. (2003). Modellingspace: Interaction design and architecture of a collaborative modelling environment. In *Proceedings of the 6th International Conference on Computer Based Learning in Science (CBLIS)* (p. 993-1004). (Retrieved April 12, 2005, from `http://www.ee.upatras.gr/hci/papers/C66_Avouris_Margaritis_Komis_Saez_M%elendez_CBLIS_2003.pdf`)

Baker, M., & Lund, K. (1997). Promoting reflective interactions in a computer-supported collaborative learning environment. *Journal of Computer Assisted Learning*, *13*(3), 175-193.

Baloian, N., Breuer, H., Hoppe, H. U., & Pino, J. A. (2004). Collaborative learning in distributed seismography. In Y. B. Kafai, W. A. Sandoval, N. Enyedy, A. Scott Nixon, & F. Herrera (Eds.), *Embracing Diversity in the Learning Sciences: Proceedings of the 6th International Conference of the Learning Sciences (ICLS)* (p. 584). Mahwah, NJ (USA): Lawrence Erlbaum.

Baloian, N., Pino, J. A., & Motelet, O. (2003). Collaborative authoring, use and reuse of learning material in a computer-integrated classroom. In J. Favela & D. Decouchant (Eds.), *Lecture Notes in Computer Science: Proceedings of the 9th International Workshop on Groupware (CRIWG)* (p. 199-207). Berlin, Germany: Springer.

Bandura, A. (1977). *Social learning theory.* Englewood Cliffs, NJ (USA): Prentice Hall.

Berge, C. (1976). *Graphs and hypergraphs.* Amsterdam, The Netherlands: North-Holland Publication Company.

Big Faceless Java Graph Library. (n.d.). Last visited april 12, 2005, at `http://big.faceless.org/products/graph`.

Biswas, G., Schwartz, D., & Bransford, J. (2001). Technology support for complex problem solving: From SAD environments to AI. In D. Forbus & P. Feltovich (Eds.), *Smart machines in education* (p. 71-97). Menlo Park, CA (USA): AAAI Press.

Bollen, L., Eimler, S., & Hoppe, H. U. (2004). SMS-based discussions technology enhanced collaboration for a literature course. In J. Roschelle, T.-W. Chan, Kinshuk, & S. J. H. Yang (Eds.), *Proceedings of the 2nd IEEE International Workshop on Wireless and Mobile Technologies in Education (WMTE)* (p. 209-210). Los Alamitos, CA (USA): IEEE Press.

Bollen, L., Hoppe, H. U., Milrad, M., & Pinkwart, N. (2002). Collaborative modelling in group learning environments. In P. I. Davidsen, E. Mollona, V. G. Diker, R. S. Langer, & J. I. Rowe (Eds.), *Proceedings of the 20th International Conference of the System Dynamics Society* (p. 53). Palermo, Italy: System Dynamics Society.

Bonk, C., & Cunningham, D. (1998). Searching for learner-centered, constructivist, and sociocultural components of collaborative educational tools. In C. Bonk & K. King (Eds.), *Electronic collaborators* (p. 25-50). Mahwah, NJ (USA): Lawrence Erlbaum.

Booch, G., Jacobson, I., & Rumbaugh, J. (1998). *The unified modeling language user guide.* Boston, MA (USA): Addison Wesley Professional.

Borghoff, U. M., & Schlichter, J. H. (1998). *Rechnergestützte Gruppenarbeit (computer supported group work).* Berlin, Germany: Springer.

Bottoni, P., Costabile, M. F., Levialdi, S., & Mussio, P. (1998). Specification of visual languages as means for interaction. In K. Marriott & B. Meyer (Eds.), *Visual language theory* (p. 353-375). Berlin, Germany: Springer.

Brandenburg, F. J. (1988). On polynomial time graph grammars. In R. Cori & M. Wirsing (Eds.), *Lecture Notes in Computer Science: Proceedings of the 5th International Conference on Theoretical Aspects of Computer Science* (p. 227-236). Berlin, Germany: Springer.

Bredeweg, B., & Forbus, K. (2003). Qualitative modeling in education. *AI Magazine*, *24*(4), 35-46.

Brooks, C., Lee, E. A., Liu, X., Neuendorffer, S., Zhao, Y., & Zheng, H. (2003). *Heterogeneous concurrent modeling and design in java (volume 1: Introduction to Ptolemy II). technical memorandum UCB/ERL M03/27* (Tech. Rep.). University of California at Berkeley (CA), USA.

BSCW homepage. (n.d.). Last visited april 12, 2005, at `http://bscw.fit.fraunhofer.de`.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., Sommerlad, P., & Stal, M. (1996). *Pattern-oriented software architecture, volume 1: A system of patterns.* Chichester, England: John Wiley & Sons.

Buzan, T. (2002). *How to mind map.* London, England: Thorsons/HarperCollins.

C# Corner. (n.d.). Last visited april 12, 2005, at `http://www.c-sharpcorner.com/Graphics.asp`.

Carriero, N., & Gelernter, D. (1989). Linda in context. *Communications of the ACM*, *32*(4), 444-458.

Celms, E., Kalnins, A., & Lace, L. (2003). Diagram definition facilities based on metamodel mappings. In J.-P. Tolvanen, J. Gray, & M. Rossi (Eds.), *Computer Science and Information Systems Reports TR-28: Proceedings of the 3rd OOPSLA Workshop on Domain-Specific Modeling* (p. 25-34). Jyväskylä, Finland: University of Jyväskylä Printing House.

Chen, P. P.-S. (1976). The entity-relationship model - toward a unified view of data. *ACM Transactions on Database Systems*, *1*(1), 9-36.

Chomsky, N. (1959). On certain formal properties of grammars. *Information and Control*, *2*, 137-167.

Cicognani, A. (2000). Concept mapping as a collaborative tool for enhanced online learning. *Educational Technology & Society*, *3*(3), 150-158.

Co-Lab project homepage. (n.d.). Last visited april 12, 2005, at `http://colab.edte.utwente.nl`.

COLDEX project homepage. (n.d.). Last visited april 12, 2005, at `http://www.coldex.info`.

Collide JGraph Howto. (n.d.). Retrieved april 12, 2005, at `http://www.collide.info/howtos/jgraph_howto/`.

Constantino-Gonzáles, M., & Suthers, D. D. (2001). Coaching web-based collaborative learning based on problem solution differences and participation. In J. D. Moore, C. L. Redfield, & W. L. Johnson (Eds.), *AI-ED in the Wired and Wireless Future: Proceedings of the 10th International Conference on Artificial Intelligence in Education (AI-ED)* (p. 176-187). Amsterdam, The Netherlands: IOS Press.

CORBA specification. (n.d.). Retrieved april 12, 2005, at `http://www.omg.org/technology/documents/corba_spec_catalog.htm`.

Cordella, L. P., Foggia, P., Sansone, C., & Vento, M. (1998). Subgraph transformations for the inexact matching of attributed relational graphs. In J.-M. Jolion & W. G. Kropatsch (Eds.), *Graph based representations in pattern recognition* (p. 43-52). Wien, Austria: Springer.

Costagliola, G., Delucia, A., Orefice, S., & Polese, G. (2002). A classification framework to support the design of visual languages. *Journal of Visual Languages and Computing*, *13*, 573-600.

Coulouris, G., Dollimore, J., & Kindberg, T. (2000). *Distributed systems: Concepts and design.* Boston, MA (USA): Addison-Wesley.

Coyle, F. P. (2002). *XML, web services, and the data revolution.* Boston, MA (USA): Addison-Wesley.

Cristani, M., & Cohn, A. G. (2002). SpaceML: A mark-up language for spatial knowledge. *Journal of Visual Languages and Computing*, *13*, 97-116.

Dabbagh, N. (2001). Concept mapping as a mindtool for critical thinking. *Journal of Computing in Teacher Education*, *17*(2), 16-24.

DATsys framework. (n.d.). Last visited april 12, 2005, at `http://www.cs.nott.ac.uk/~azt/research.htm`.

DCOM technical overview. (n.d.). Last visited april 12, 2005, at `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndcom%/html/msdn_dcomtec.asp`.

DeGroot, M. H. (1989). *Probability and stochastics.* Boston, MA (USA): Addison Wesley.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, *1*, 269-271.

Dillenbourg, P. (1999). What do you mean by "collaborative learning"? In P. Dillenbourg (Ed.), *Collaborative learning: Cognitive and computational approaches* (p. 1-19). Amsterdam, The Netherlands: Pergamon.

Dillenbourg, P., & Self, J. (1995). Designing human-computer collaborative learning. In C. O'Malley (Ed.), *Computer-supported collaborative learning* (p. 245-264). Berlin, Germany: Springer.

Dinesh, T. B., & Üsküdarlı, S. (1998). Input and output for specified visual languages. In K. Marriott & B. Meyer (Eds.), *Visual language theory* (p. 325-351). Berlin, Germany: Springer.

Distributed systems technology centre. (n.d.). Last visited april 12, 2005, at `http://www.dstc.edu.au`.

Dix, A., Finlay, J., Abowd, G. D., & Beale, R. (2004). *Human-computer interaction.* Harlow, England: Pearson Education Limited.

Dolk, D. R., & Kottemann, J. E. (1993). Model integration and a theory of models. *Decision Suupport Systems*, *9*, 51-63.

DOME Guide. (1999). Retrieved april 12, 2005, at `http://www.htc.honeywell.com/dome/DOMEGuide.pdf`.

DOME homepage. (n.d.). Last visited april 12, 2005, at `http://www.htc.honeywell.com/dome`.

Endsley, M. (1995). Toward a theory of situation awareness in dynamic systems. *Human Factors*, *37*(1), 32-64.

Ferrucci, F., Tortora, G., Tucci, M., & Vitiello, G. (1998). Relation grammars: A formalism for syntactic and semantic analysis of visual languages. In K. Marriott & B. Meyer (Eds.), *Visual language theory* (p. 219-243). Berlin, Germany: Springer.

Flenner, R. (2001). *Jini and JavaSpaces application development.* Indianapolis, IN (USA): Sams.

Ford, L. R., & Fulkerson, D. R. (1962). *Flows in networks.* Princeton, NJ (USA): Princeton University Press.

Forrester, J. W. (1968). *Principles of systems.* Waltham, MA (USA): Pegasus Communications.

Fosnot, C. (1996). *Constructivism: theory, perspectives, and practice.* New York, NY (USA): Teachers College Press.

Freeman, E., Hupfer, S., & Arnold, K. (1999). *JavaSpaces principles, patterns, and practice.* Boston (MA), USA: Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns. elements of reusable object-oriented software.* Boston, MA (USA): Addison-Wesley Professional.

Gansner, E. R., & North, S. C. (2000). An open graph visualization system and its applications to software engineering. *Software - Practice and Experience*, *30*(11), 1203-1233.

Gaßner, K. (2003). *Diskussionen als Szenario zur Ko-Konstruktion von Wissen mit visuellen Sprachen (Using the discussion scenario for the co-construction of knowledge with visual languages).* Published online at `http://www.ub.uni-duisburg.de/ETD-db`. (Dissertation at the University of Duisburg-Essen, Germany)

Gaßner, K., Jansen, M., Harrer, A., Herrmann, K., & Hoppe, H. U. (2003). Analysis methods for collaborative models and activities. In B. Wasson, S. Ludvigsen, & H. U. Hoppe (Eds.), *Designing for Change in Networked Learning Environments: Proceedings of the 5th International Conference on Computer Support for Collaborative Learning (CSCL)* (p. 411-420). Dordrecht, The Netherlands: Kluwer Academic Publishers.

Gaßner, K., Tewissen, F., Mühlenbrock, M., Loesch, A., & Hoppe, H. U. (1998). Intelligently supported collaborative learning environments based on visual languages: a generic approach. In F. Darses & P. Zaraté (Eds.), *Proceedings of the 3rd International Conference on the Design of Cooperative Systems (COOP)* (p. 47-55). Sophia Antipolis, France: INRIA.

Geoffrion, A. (1987). An introduction to structured modeling. *Management Science*, *33*(5), 547-588.

Geoffrion, A. (1989a). The formal aspects of structured modeling. *Operations Research*, *37*(1), 30-51.

Geoffrion, A. (1989b). Integrated modeling systems. *Computer Science in Economics and Management*, *2*(1), 3-15.

GigaSpaces Homepage. (n.d.). Last visited april 12, 2005, at `http://www.gigaspaces.com`.

Glasersfeld, E. von. (1995). *Radical concstructivism - a way of knowing and learning.* London, England: Falmer Press.

Goldman, S. V. (1996). Mediating microworlds: Collaboration on high school science activities. In T. Koschmann (Ed.), *CSCL: Theory and practice of an emerging paradigm* (p. 45-81). New York, NY (USA): Lawrence Erlbaum.

Gredler, M. E. (1992). *Learning and instruction - theory into practice.* New York, NY (USA): Macmillan Publishing Company.

Gross, J., & Yellen, J. (1999). *Graph theory and its applications.* Boca Raton, FL (USA): CRC Press.

Gutwin, C., & Greenberg, S. (2004). The importance of awareness for team cognition in distributed collaboration. In E. Salas & S. M. Fiore (Eds.), *Team cognition: Understanding the factors that drive process and performance* (p. 177-201). Washington, DC (USA): APA Press.

Haarslev, V. (1999). A logic-based formalism for reasoning about visual representations. *Journal of Visual Languages and Computing*, *10*(4), 421-445.

Harel, D., & Naamad, A. (1996). The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, *5*(4), 293-333.

Harel, D., & Rumpe, B. (2004). Meaningful modeling: What's the semantics of "semantics"? *Computer*, *37*(10), 64-72.

Heiler, S. (1995). Semantic interoperability. *ACM Computing Surveys*, *27*(2), 271-273.

Herrmann, K., Hoppe, H. U., & Pinkwart, N. (2003). A checking mechanism for visual language environments. In H. U. Hoppe, F. Verdejo, & J. Kay (Eds.), *Shaping the Future of Learning through Intelligent Technologies: Proceedings of the 11th International Conference on Artificial Intelligence in Education (AI-ED)* (p. 97-104). Amsterdam, The Netherlands: IOS Press.

Herrmann, T. (2001). Kommunikation und Kooperation (communication and cooperation). In G. Schwabe, N. Streitz, & R. Unland (Eds.), *CSCW-Kompendium* (p. 15-32). Berlin, Germany: Springer.

Hoeksema, K., Jansen, M., & Hoppe, H. U. (2004). Interactive processing of astronomical observations in a cooperative modelling environment. In Kinshuk, C.-K. Looi, E. Sutinen, D. Sampson, I. Aedo, L. Uden, & E. Kähkönen (Eds.), *Proceedings of the 4th IEEE International Conference on Advanced Learning Technologies (ICALT)* (p. 888-889). Los Alamitos, CA (USA): IEEE Press.

Honebein, P. C. (1996). Seven goals for the design of constructivist learning environments. In B. G. Wilson (Ed.), *Constructivist learning environments: Case studies in instructional design* (p. 11-24). Englewood Cliffs, NJ (USA): Educational Technology Publications.

Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2000). *Introduction to automata theory, languages, and computation.* Boston, MA (USA): Addison Wesley.

Hoppe, H. U. (1995). The use of multiple student modeling to parameterize group learning. In J. Greer (Ed.), *Proceedings of the 7th World Conference on Artificial Intelligence in Education (AI-ED)* (p. 234-241). Charlottesville, VA (USA): AACE.

Hoppe, H. U. (2001). Collaborative mind tools for the classroom: Strategies for pedagogical innovation. Keynote address. In *Proceedings of the 9th International Conference on Computers in Education (ICCE).* (Presentation slides available at `http://www.collide.info/Projects/seed/Icce2001_presentation/index.htm`. Last visited April 12, 2005.)

Hoppe, H. U. (2002). Computers in the classroom - a disappearing phenomenon? In A. Dimitracopoulou (Ed.), *Proceedings of the 3rd Hellenic Conference with International Participation on Information and Communication Technologies in Education (HICTE)* (p. 19-30). Rhodes, Greece: KASTANIOTIS Editions - Inter@ctive.

Hoppe, H. U. (2004). Collaborative mind tools. In M. Tokoro & L. Steels (Eds.), *A learning zone of one's own - sharing representations and flow in collaborative learning environments* (p. 223-234). Amsterdam, The Netherlands: IOS Press.

Hoppe, H. U. (2005). Educational information technologies and collaborative learning - anything new? In H. U. Hoppe, A. Soller, & H. Ogata (Eds.), *New technologies for collaborative learning (to appear).*

Hoppe, H. U., Gaßner, K., Mühlenbrock, M., & Tewissen, F. (2000). Distributed visual language environments for cooperation and learning: Applications and intelligent support. *Group Decision and Negotiation*, *9*(3), 205-220.

Hoppe, H. U., & R-Plötzner. (1999). Can analytic models support learning in groups? In P. Dillenbourg (Ed.), *Collaborative learning: Cognitive and computational approaches* (p. 147-168). Amsterdam, The Netherlands: Pergamon.

Hübscher-Younger, T., & Narayanan, N. H. (2003). Designing for divergence. In B. Wasson, S. Ludvigsen, & H. U. Hoppe (Eds.), *Designing for Change in Networked Learning Environments: Proceedings of the 5th International Conference on Computer Support for Collaborative Learning (CSCL)* (p. 461-470). Dordrecht, The Netherlands: Kluwer Academic Publishers.

Hylands, C., Lee, E. A., Liu, J., Liu, X., Neuendorffer, S., Xiong, Y., Zhao, Y., & Zheng, H. (2003). *Overview of the Ptolemy project. technical memorandum UCB/ERL M03/25* (Tech. Rep.). University of California at Berkeley (CA), USA.

Ikeda, M., Go, S., & Mizoguchi, R. (1997). Opportunistic group formation. In B. du Bulay & R. Mizoguchi (Eds.), *Knowledge and Media in Learning Systems: Proceedings of the 8th International Conference on Artificial Intelligence in Education (AI-ED)* (p. 167-174). Amsterdam, The Netherlands: IOS Press.

Jackson, S. L., Stratford, S. J., Krajcik, J. S., & Soloway, E. (1996). Making dynamic modeling accessible to pre-college science students. *Interactive Learning Environments*, *4*(3), 233-257.

Jansen, M. (2003). MatchMaker TNG - a framework to support collaborative java applications. In H. U. Hoppe, F. Verdejo, & J. Kay (Eds.), *Shaping the Future of Learning through Intelligent Technologies: Proceedings of the 11th International Conference on Artificial Intelligence in Education (AI-ED)* (p. 529-530). Amsterdam, The Netherlands: IOS Press.

Jansen, M., Oelinger, M., Hoeksema, K., & Hoppe, H. U. (2004). Exploring the use of mobile devices to facilitate educational interoperability around digitally enhanced experiments. In J. Roschelle, T.-W. Chan, Kinshuk, & S. J. H. Yang (Eds.), *Proceedings of the 2nd IEEE International Workshop on Wireless and Mobile Technologies in Education (WMTE)* (p. 83-90). Los Alamitos, CA (USA): IEEE Press.

Jansen, M., Pinkwart, N., & Tewissen, F. (2001). MatchMaker - Flexible Synchronisation von Java-Anwendungen (MatchMaker - flexible synchronization of Java applications). In R. Klinkenberg, S. Rüping, A. Fick, N. Henze, C. Herzog, R. Molitor, & O. Schröder (Eds.), *Forschungsbericht 763: Tagungsband der GI-Workshopwoche "Lernen-Lehren-Wissen-Adaptivität"* (p. 180-186). Dortmund, Germany: University of Dortmund.

Java Foundation Classes Homepage. (n.d.). Last visited april 12, 2005, at `http://java.sun.com/products/jfc`.

Java Imaging and Graphics Library. (n.d.). Last visited april 12, 2005, at `http://rivit.cs.byu.edu/jigl/`.

Java Open Source Graph Visualization Component Suite. (n.d.). Last visited april 12, 2005, at `http://www.jgraph.com`.

Java Shared Data Toolkit Homepage. (n.d.). Last visited april 12, 2005, at `https://jsdt.dev.java.net/`.

Jensen, N., Seipel, S., Nejdl, W., & Olbrich, S. (2003). CoVASE: Collaborative visualization for constructivist learning. In B. Wasson, S. Ludvigsen, & H. U. Hoppe (Eds.), *Designing for Change in Networked Learning Environments: Proceedings of the 5th International Conference on Computer Support for Collaborative Learning (CSCL)* (p. 249-253). Dordrecht, The Netherlands: Kluwer Academic Publishers.

Jermann, P., Soller, A., & Mühlenbrock, M. (2001). From mirroring to guiding: A review on state of the art technology for supporting collaborative learning. In P. Dillenbourg, A. Eurelings, & K. Hakkarainen (Eds.), *Proceedings of the European Conference on Computer-Supported Collaborative Learning (Euro-CSCL)* (p. 324-331). Maastricht, The Netherlands: McLuhan Institute.

Johnson, D. W., & Johnson, R. T. (1990). Co-operative learning and achievement. In S. Sharan (Ed.), *Co-operative learning: Theory and research* (p. 23-37). New York, NY (USA): Praeger.

Jonassen, D. H. (2000). *Computers as mindtools for schools.* Upper Saddle River, NJ (USA): Prentice Hall.

Jonassen, D. H., Peck, K. L., & Wilson, B. G. (1999). *Learning with technology: A constructivist perspective.* Columbus, OH (USA): Prentice Hall.

Joolingen, W. R. van. (2000). Designing for collaborative discovery learning. In G. Gauthier, C. Frasson, & K. VanLehn (Eds.), *Lecture Notes in Computer Science: Proceedings of the 5th International Conference on Intelligent Tutoring Systems (ITS)* (p. 202-211). Berlin, Germany: Springer.

Joolingen, W. R. van, King, S., & Jong, T. de. (1997). The simquest authoring system for simulation-based discovery environments. In B. du Bulay & R. Mizoguchi (Eds.), *Knowledge and media in learning spaces* (p. 79-87). Amsterdam, The Netherlands: IOS Press.

Joolingen, W. R. van, & Löhner, S. (2001). Representations in collaborative modeling tasks. In *Proceedings of the workshop on "External representations in AIED: Multiple forms and multiple roles" at the 10th International Conference on Artificial Intelligence in Education (AI-ED).* (Retrieved April 12, 2005, from `http://www.psychology.nottingham.ac.uk/research/credit/AIED-ER/vanjooli%ngen.pdf`)

JUNG manual. (n.d.). Retrieved april 12, 2005, at `http://jung.sourceforge.net/doc/manual.html`.

Kaul, M. (1982). Parsing of graphs in linear time. In H. Ehrig, M. Nagl, & G. Rozenberg (Eds.), *Lecture Notes in Computer Science: Proceedings of the 2nd International Workshop on Graph Grammars and their Application in Computer Science* (p. 206-218). Berlin, Germany: Springer.

Kay, A., & Goldberg, A. (1977/2001). Personal dynamic media. In R. Packer & K. Jordan (Eds.), *multimedia - from wagner to virtual reality* (p. 167-178). London, England: W.W. Norton & Company.

Koedinger, K. R., Suthers, D. D., & Forbus, K. D. (1999). Component-based construction of a science learning space. *International Journal of Artificial Intelligence in Education*, *10*, 292-313.

Koschmann, T. (2002). Dewey's contribution to the foundations of CSCL research. In G. Stahl (Ed.), *Foundations for a CSCL Community: Proceedings of the 4th International Conference on Computer Support for Collaborative Learning (CSCL)* (p. 17-22). Hillsdale, NJ (USA): Lawrence Erlbaum.

Kuan, C. L., Lee, C. S., & Ho, C. K. (2003). Agent-assisted collaborative concept map. In V. Devedzic, J. M. Spector, D. G. Sampson, & Kinshuk (Eds.), *Proceedings of the 3rd IEEE International Conference on Advanced Learning Technologies (ICALT)* (p. 282-283). Los Alamitos, CA (USA): IEEE Press.

Kuhn, M., Hoppe, U., Lingnau, A., & Fendrich, M. (2004). Evaluation of exploratory approaches in learning probability based on computational modelling and simulation. In P. Isaias, Kinshuk, & D. G. Sampson (Eds.), *Proceedings of the IADIS conference of Cognition and Exploratory Learning in Digital Age (CELDA)* (p. 83-90). Lisbon, Portugal: IADIS Press.

Kuhn, M., Jansen, M., Harrer, A., & Hoppe, H. U. (2005). A lightweight approach for flexible group management in the classroom. In T. Koschmann, D. Suthers, & T.-W. Chan (Eds.), *Computer Supported Collaborative Learning 2005 - The Next 10 Years! Proceedings of the 6th International Conference on Computer Support for Collaborative Learning (CSCL)* (p. 353-357). Mahwah (NJ), USA: Lawrence Erlbaum Associates.

Kurtz dos Santos, A. D. C., & Ogborn, J. (1994). Sixth form student's ability to engage in computational modelling. *Journal of Computer Assisted Learning*, *10*(3), 182-200.

Kynigos, C. (2002). Generating cultures for mathematical microworld development in a multi-organizational context. *Journal of Educational Computing Research*, *1+2*, 183-209.

Lara, J. de, & Vangheluwe, H. (2004). Defining visual notations and their manipulations through meta-modeling and graph transformation. *Journal of Visual Languages and Computing*, *15*, 309-330.

Larkin, J. H., & Simon, H. A. (1987). Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, *11*, 65-99.

Lauer, M., Ueberall, M., Horvath, O., Matthes, M., & Drobnik, O. (2003). CLE: A collaborative learning environment. In B. Wasson, R. Baggetun, H. U. Hoppe, & S. Ludvigsen (Eds.), *Community Events: Communication and Interaction. Proceedings of the 5th International Conference on Computer Support for Collaborative Learning (CSCL)* (p. 120-122). Bergen, Norway: Intermedia.

Law, N., & Tam, E. (1998). WORLDMAKER (HK) - an iconic modelling tool for children to explore complex behaviour. In T. W. Chan, A. Collins, & J. Lin (Eds.), *Global Education on the Net: Proceedings of the 6th International Conference on Computers in Education (ICCE)* (p. 466-472). Berlin, Germany: Springer.

Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., & Volgyesi, P. (2001). The generic modeling environment. In *Proceedings of the IEEE International Workshop on Intelligent Signal Processing (WISP)*. (Retrieved April 12, 2005, from `http://www.isis.vanderbilt.edu/Projects/gme/GME2000Overview.pdf`)

Lego Mindstorms Homepage. (n.d.). Last visited april 12, 2005, at `http://mindstorms.lego.com`.

Lenard, M. L. (1993). A prototype implementation of a model management system for discrete-event simulation models. In G. W. Evans, M. Mollaghasemi, E. C. Russell, & W. E. Biles (Eds.), *Proceedings of the 25th conference on Winter simulation* (p. 560 - 568). New York, NY (USA): ACM Press.

Liebold, R., & Trinczek, R. (2002). Experteninterview (expert interview). In S. Kühn & P. Strodtholz (Eds.), *Methoden der Organisationsforschung* (p. 33-70). Reinbeck, Germany: Rowohlt Taschenbuch Verlag.

Ligozat, G. (1998). Reasoning about cardinal directions. *Journal of Visual Languages and Computing*, *9*(1), 23-44.

Linden, J. L. v. d., Erkens, G., Schmidt, H., & Renshaw, P. (2000). Collaborative learning. In P. R. J. Simons, J. L. van der Linden, & T. M. Duffy (Eds.), *New learning* (p. 37-55). Dordrecht, The Netherlands: Kluwer Academic Publishers.

Lingnau, A., Kuhn, M., Harrer, A., Hofmann, D., Fendrich, M., & Hoppe, H. U. (2003). Enriching traditional classroom scenarios by seamless integration of interactive media. In V. Devedzic, J. M. Spector, D. G. Sampson, & Kinshuk (Eds.), *Proceedings of the 3rd IEEE International Conference on Advanced Learning Technologies (ICALT)* (p. 135-139). Los Alamitos, CA (USA): IEEE Press.

Littleton, K., & Häkkinen, P. (1999). Learning together: Understanding the process of computer-based collaborative learning. In P. Dillenbourg (Ed.), *Collaborative learning: Cognitive and computational approaches* (p. 20-30). Amsterdam, The Netherlands: Pergamon.

Löhner, S., Joolingen, W. R. van, & Savelsbergh, E. R. (2003). The effect of external representation on constructing computer models of complex phenomena. *Instructional Science*, *31*, 395-418.

Luchini, K., Quintana, C., & Soloway, E. (2003). Pocket PiCoMap: a case study in designing and assessing a handheld concept mapping tool for learners. In G. Cockton & P. Korhonen (Eds.), *Proceedings of the conference on Human Factors in Computing Systems* (p. 321-328). New York (NY), USA: ACM Press.

MacLean, A., Young, R. M., Bellotti, V., & Moran, T. (1991). Questions, options, and criteria: elements of design space analysis. *Human-Computer-Interaction*, *6*(3&4), 201-250.

Margaritis, M., Fidas, C., Avouris, N., & Komis, V. (2003). A peer-to-peer architecture for synchronous collaboration over low-bandwidth networks. In K. Margaritis & I. Pitas (Eds.), *Proceedings of the 9th Panhellenic Conference in Informatics* (p. 231-242). (Retrieved April 12, 2005, from `http://www.ee.upatras.gr/hci/papers/C74_Margaritis_etal_EPY9_v03.pdf`)

Marriott, K., & Meyer, B. (1998). The CCMG visual language hierarchy. In K. Marriott & B. Meyer (Eds.), *Visual language theory* (p. 129-169). Berlin, Germany: Springer.

Marriott, K., Meyer, B., & Wittenburg, K. B. (1998). A survey of visual language specification and recognition. In K. Marriott & B. Meyer (Eds.), *Visual language theory* (p. 5-85). Berlin, Germany: Springer.

McBrien, P., & Poulovassilis, A. (1999). A uniform approach to inter-model transformations. In M. Jarke & A. Oberweis (Eds.), *Lecture Notes in Computer Science: Proceedings of the 11th International Conference on Advanced Information Systems Engineering (CAiSE)* (p. 333-348). Berlin, Germany: Springer.

McLaren, B., Bollen, L., Walker, E., Harrer, A., & Sewall, J. (2005). Cognitive tutoring of collaboration: Developmental and empirical steps towards realization. In T. Koschmann, D. Suthers, & T.-W. Chan (Eds.), *Computer Supported Collaborative Learning 2005 - The Next 10 Years! Proceedings of the 6th International Conference on Computer Support for Collaborative Learning (CSCL)* (p. 418-422). Mahwah (NJ), USA: Lawrence Erlbaum Associates.

Merton, R. K., & Kendall, P. L. (1946). The focused interview. *American Jounal of Sociology, 51*, 541-557.

Meta-Object Facility Specification. (n.d.). Retrieved august 21, 2004, at `http://www.omg.org/cwm`.

Metacase technology whitepaper: Domain-specific modeling - 10 times faster than UML. (n.d.). Retrieved april 12, 2005, at `http://www.metacase.com/papers/Domain-specific_modeling_10X_faster_than%_UML.pdf`.

Meyer, B. (1994). *Visuelle logische Sprachen zur Behandlung räumlicher Information (Visual logic languages for spatial information handling).* Dissertation at the FernUni Hagen, Germany.

Meyers Enzyklopädisches Lexikon (meyers encyclopedic lexicon). (1976). Mannheim, Germany: Lexikonverlag Bibliographisches Institut.

Microsoft Office Homepage. (n.d.). Last visited april 12, 2005, at `http://office.microsoft.com/en-us/default.aspx`.

Microsoft Windows Server: Making collaboration the engine of team productivity. (2004). Retrieved april 12, 2005, at `http://www.microsoft.com/windowsserver2003/techinfo/overview/WSSvision.%mspx`.

Milrad, M., Hoppe, H. U., Gottdenker, J., & Jansen, M. (2004). Exploring the use of mobile devices to facilitate educational interoperability around digitally enhanced experiments. In J. Roschelle, T.-W. Chan, Kinshuk, & S. J. H. Yang (Eds.), *Proceedings of the 2nd IEEE International Workshop on Wireless and Mobile Technologies in Education (WMTE)* (p. 182-186). Los Alamitos, CA (USA): IEEE Press.

Milrad, M., Spector, J. M., & Davidsen, P. I. (2002). Model facilitated learning. In S. Naidu (Ed.), *Learning and teaching with technology: Principles and practices* (p. 13-27). London, England: Kogan Page Publishers.

ModelIt Homepage. (n.d.). Last visited april 12, 2005, at `http://www.goknow.com/Products/Model-It`.

ModellingSpace project homepage. (n.d.). Last visited march 7, 2005, at `http://www.modellingspace.net`.

Morteo, G. L., & Mariscal, G. L. (2003). An electronic ludic learning environment for mathematics based on learning objects. In D. Lassner & C. McNaught (Eds.), *Proceedings of the World Conference on Educational Multimedia, Hypermedia and Telecommunications (ED-MEDIA)* (p. 849-852). Norfolk, VA (USA): AACE.

Mühlenbrock, M. (2001). *Action-based collaboration analysis for group learning.* Amsterdam, The Netherlands: IOS Press.

Mühlenbrock, M., Tewissen, F., & Hoppe, H. U. (1997). A framework system for intelligent support in open distributed learning environments. In B. du Bulay & R. Mizoguchi (Eds.), *Knowledge and Media in Learning Systems: Proceedings of the 8th International Conference on Artificial Intelligence in Education (AI-ED)* (p. 191-198). Amsterdam, The Netherlands: IOS Press.

Mwakitalima, R. (2003). *Asserting integrity in synchronous communication architectures.* Unpublished master's thesis, University of Duisburg-Essen.

NetMeeting Resource Kit. (n.d.). Last visited april 12, 2005, at `http://www.microsoft.com/technet/prodtechnol/netmting/reskit/netmtg3/de%fault.mspx`.

Novak, J. D., & Gowin, D. B. (1984). *Learning how to learn.* Cambridge, England: Cambridge University Press.

OpenJGraph Homepage. (n.d.). Last visited april 12, 2005, at `http://openjgraph.sourceforge.net`.

Or-Bach, R. (2003). Design consideration for supporting collaborative modeling. In V. Devedzic, J. M. Spector, D. G. Sampson, & Kinshuk (Eds.), *Proceedings of the 3rd IEEE International Conference on Advanced Learning Technologies (ICALT)* (p. 219-223). Los Alamitos, CA (USA): IEEE Press.

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas.* New York, NY (USA): Basic Books.

Perkins, D. (1991). Technology meets constructivism: Do they make a marriage? *Educational Technology, 31*(5), 18-23.

Petri, C. A. (1962). *Kommunikation mit Automaten (communication with automata).* Bonn, Germany: Schriften des Rheinisch-Westfälischen Instituts für Instrumentelle Mathematik.

Pinkwart, N. (2003). A plug-in architecture for graph based collaborative modeling systems. In H. U. Hoppe, F. Verdejo, & J. Kay (Eds.), *Shaping the Future of Learning through Intelligent Technologies: Proceedings of the 11th International Conference on Artificial Intelligence in Education (AI-ED)* (p. 535-536). Amsterdam, The Netherlands: IOS Press.

Pinkwart, N., Hoppe, H. U., Bollen, L., & Fuhlrott, E. (2002). Group-oriented modelling tools with heterogeneous semantics. In S. A. Cerri, G. Gouardères, & F. Paraguaçu (Eds.), *Lecture Notes in Computer Science: Proceedings of the 6th International Conference on Intelligent Tutoring Systems (ITS)* (p. 21-30). Berlin, Germany: Springer.

Pinkwart, N., Hoppe, H. U., & Gaßner, K. (2001). Integration of domain-specific elements into visual language based collaborative environments. In M. R. S. Borges, J. M. Haake, & H. U. Hoppe (Eds.), *Proceedings of the 7th International Workshop on Groupware (CRIWG)* (p. 142-147). Los Alamitos, CA (USA): IEEE Press.

Pinkwart, N., Jansen, M., Oelinger, M., Korchounova, L., & Hoppe, U. (2004). Partial generation of contextualized metadata in a collaborative modeling environment. In L. Aroyo & C. Tasso (Eds.), *Workshop proceedings of the 3rd International Conference on Adaptive Hypermedia (AH)* (p. 372-376). Eindhoven, The Netherlands: Technische Universiteit Eindhoven.

Ptolemy project homepage. (n.d.). Last visited april 12, 2005, at `http://ptolemy.berkeley.edu`.

Randell, D. A., Cui, Z., & Cohn, A. G. (1992). A spatial logic based on regions and connection. In B. Swartout & B. Nebel (Eds.), *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning* (p. 165-176). Los Altos, CA (USA): Morgan Kaufmann.

Read, T., Verdejo, F., & Barros, B. (2003). Incorporating interoperability into a distributed elearning system. In D. Lassner & C. McNaught (Eds.), *Proceedings of the World Conference on Educational Multimedia, Hypermedia and Telecommunications (ED-MEDIA)* (p. 273-282). Norfolk, VA (USA): AACE.

Rekers, J., & Schürr, A. (1997). Defining and parsing visual languages with layered graph grammars. *Journal of Visual Languages and Computing*, *8*, 27-55.

Repenning, A. (1994). Programming substrates to create interactive learning environments. *Journal of Interactive Learning Environments*, *4*(1), 45-74.

Roschelle, J., DiGiano, C., & Chung, M. (2000). Reusability and interoperability of tools for mathematics learning: Lessons from the ESCOT project. In F. Haghdy & F. Kurfess (Eds.), *Proceedings of the International Congress on Intelligent Systems and Applications* (p. 664-669). Wetaskiwin, Canada: ICSC Academic Press.

Roschelle, J., DiGiano, C., Repenning, A., Phillips, J., Jackiw, N., & Suthers, D. (1999). Developing educational software components. *Computer*, *32*(9), 50-58.

Roschelle, J., & Teasley, S. D. (1995). The construction of shared knowledge in collaborative problem solving. In C. O'Malley (Ed.), *Computer supported collaborative learning* (p. 69-96). Berlin, Germany: Springer.

Savage, T., Sanchez, I. A., O'Donnel, F., & Tangney, B. (2003). *Using robotic technology as a constructivist mindtool in knowledge construction.* Los Alamitos, CA (USA): IEEE Press.

Schümmer, J., & Schuckmann, C. (2001). Synchrone Softwarearchitekturen (synchronous software architectures). In G. Schwabe, N. Streitz, & R. Unland (Eds.), *CSCW-Kompendium* (p. 297-309). Berlin, Germany: Springer.

Schunk, D. H. (1991). *Learning theories: An educational perspective.* New York, NY (USA): Macmillan Publishing Company.

SEED project homepage. (n.d.). Last visited april 12, 2005, at `http://ilios.cti.gr/seed`.

Senge, P. M. (1990). *The fifth discipline: The art and practice of the learning organization.* New York, NY (USA): Doubleday Books.

Sfard, A. (1998). On two metaphors for learning and the dangers of choosing just one. *Educational Researcher*, *27*(2), 4-13.

Shneiderman, B. (1983). Direct manipulation. a step beyond programming languages. *IEEE Transactions on Computers*, *16*(8), 57-69.

Sierhuis, M. (2001). *Modeling and simulating work practice. BRAHMS: a multiagent modeling and simulation language for work system analysis and design.* Amsterdam, The Netherlands: SIKS Dissertation Series.

Sierhuis, M., & Selvin, A. M. (1996). Towards a framework for collaborative modeling and simulation (Position paper for workshop on strategies for collaborative modeling and simulation). In M. S. Ackerman (Ed.), *Proceedings of the 5th ACM Conference on Computer Supported Cooperative Work (CSCW)* (p. 2). New York, NY (USA): ACM Press. (Retrieved April 12, 2005, from `http://www.compendiuminstitute.org/compendium/papers/SierhuisSelvin-CSC%W-1996.PDF`)

Silander, P., Sutinen, E., & Tarhio, J. (2004). Mobile collaborative concept mapping - combining classroom activity with simultaneous field exploration. In J. Roschelle, T.-W. Chan, Kinshuk, & S. J. H. Yang (Eds.), *Proceedings of the 2nd IEEE International Workshop on Wireless and Mobile Technologies in Education (WMTE)* (p. 114-118). Los Alamitos, CA (USA): IEEE Press.

Skarmeta, A. F. G., Joolingen, W. R. van, Martinez, E., Celdrán, M., & Mora, M. (2002). *Co-lab basic architecture (Co-Lab project deliverable).* Retrieved April 12, 2005, at `http://colab.edte.utwente.nl/documents/d4.pdf`.

Skiena, S. (1990). *Implementing discrete mathematics.* Reading, MA (USA): Addison-Wesley.

Skinner, B. F. (1974). *About behaviourism.* London, England: Jonathan Cape Ltd.

Soller, A., & Lesgold, A. (2003). A computational approach to analyzing online knowledge sharing interaction. In H. U. Hoppe, F. Verdejo, & J. Kay (Eds.), *Shaping the Future of Learning through Intelligent Technologies: Proceedings of the 11th International Conference on Artificial Intelligence in Education (AI-ED)* (p. 253-260). Amsterdam, The Netherlands: IOS Press.

Soloway, E., Pryor, A. Z., Krajcik, J. S., Jackson, S., Stratford, S. J., Wisnudel, M., & Klein, J. T. (1997). Scienceware's Model-It: Technology to support authentic science inquiry. *T.H.E. Journal*, *25*(3), 54-56.

Sprinkle, J., & Karsai, G. (2003). Model migration through visual modeling. In J.-P. Tolvanen, J. Gray, & M. Rossi (Eds.), *Computer Science and Information Systems Reports TR-28: Proceedings of the 3rd OOPSLA Workshop on Domain-Specific Modeling* (p. 51-58). Jyväskylä, Finland: University of Jyväskylä Printing House.

Steen, M. van, Homburg, P., & Tanenbaum, A. (1999). GLOBE: A wide-area distributed system. *IEEE Concurrency*, *7*(1), 70-78.

Stefik, M., Foster, G., Bobrow, D. G., Kahn, K., Lanning, S., & Suchman, L. (1987). Beyond the chalkboard: Computer support for collaboration and problem solving in meetings. *Communications of the ACM*, *30*(1), 32-47.

Sun, C., & Ellis, C. (1989). Operational transformation in real-time group editors: Issues, algorithms, and achievements. In S. Poltrock & J. Grudin (Eds.), *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW)* (p. 58-68). New York (NY), USA: ACM Press.

Suppes, P., & Macken, E. (1978). The historical path from research and development to operational use of CAI. *Educational Technology*, *18*(4), 9-12.

Suthers, D. D. (1999a). Effects of alternate representations of evidential relations on collaborative learning discourse. In C. Hoadley & J. Roschelle (Eds.), *Proceedings of the 3rd International Conference on Computer Support for Collaborative Learning (CSCL)* (p. 611-620). Mahwah, NJ (USA): Lawrence Erlbaum.

Suthers, D. D. (1999b). Representational bias as guidance for learning interactions: A research agenda. In S. P. Lajoie & M. Vivet (Eds.), *Frontiers in Artificial Intelligence and Applications: Proceedings of the 9th World Conference on Artificial Intelligence in Education (AI-ED)* (p. 611-620). Amsterdam, The Netherlands: IOS Press.

Suthers, D. D. (2001). Architectures for computer supported collaborative learning. In J. R. Hartley, T. Okamoto, Kinshuk, & J. P. Klus (Eds.), *Proceedings of the 2nd IEEE International Conference on Advanced Learning Technologies (ICALT)* (p. 25-28). Los Alamitos, CA (USA): IEEE Press.

Suthers, D. D., Connelly, J., Lesgold, A., Paolucci, M., Toth, E., Toth, J., & Weiner, A. (2001). Representational and advisory guidance for students learning scientific inquiry. In D. Forbus & P. Feltovich (Eds.), *Smart machines in education* (p. 7-35). Menlo Park, CA (USA): AAAI Press.

Suthers, D. D., & Dwyer, N. (2004). *personal communication.*

Suthers, D. D., & Hundhausen, C. D. (2003). An experimental study of the effects of representational guidance on collaborative learning processes. *Journal of the Learning Sciences*, *12*(2), 183-219.

Suthers, D. D., Toth, E. E., & Weiner, A. (1997). An integrated approach to implementing collaborative inquiry in the classroom. In R. Hall, N. Miyake, & N. Enyedy (Eds.), *Proceedings of the 2nd International Conference on Computer Support for Collaborative Learning (CSCL)* (p. 272-279). Mahwah, NJ (USA): Lawrence Erlbaum.

Suthers, D. D., Weiner, A., Connelly, J., & Paolucci, M. (1995). Belvedere: Engaging students in critical discussion of science and public policy issues. In J. Greer (Ed.), *Proceedings of the 7th World Conference on Artificial Intelligence in Education (AI-ED)* (p. 266-273). Charlottesville (VA), USA: Association for the Advancement of Computing in Education.

Suzuki, H., & Funaoi, H. (2002). Community incubator: Supporting construction of online learners' community through visualization. In Kinshuk, R. Lewis, K. Akahori, R. Kemp, T. Okamoto, L. Henderson, & C. H. Lee (Eds.), *Proceedings of the 10th International Conference on Computers in Education (ICCE)* (p. 399-403). Los Alamitos, CA (USA): IEEE Press.

Synergo homepage. (n.d.). Last visited april 12, 2005, at `http://www.synergo.gr`.

Sztipanovits, J., Karsai, G., Biegl, C., Bapty, T., Ledeczi, A., & Misra, A. (1995). Multigraph: an architecture for model-integrated computing. In *Proceedings of the 1st International Conference on Engineering of Complex Computer Systems* (p. 361-368). Washington (DC), USA: IEEE Computer Society.

Tanenbaum, A. S., & Steen, M. van. (2002). *Distributed systems: Principles and paradigms.* Upper Saddle River, NJ (USA): Prentice Hall.

Tewissen, F., Baloian, N., Hoppe, H. U., & Reimberg, E. (2000). "MatchMaker" synchronizing objects in replicated software-architectures. In C. Salgado (Ed.), *Proceedings of the 6th International Workshop on Groupware (CRIWG)* (p. 60-67). Washington (DC), USA: IEEE Computer Society.

Tiller, M. (2001). *Introduction to physical modeling with Modelica.* Dordrecht, The Netherlands:: Kluwer Academic Publishers.

Tolvanen, J.-P., & Kelly, S. (2004). Domänenspezifische Modellierung (Domain specific modeling). *Objektspektrum*, *4*, 30-34.

TouchGraph Homepage. (n.d.). Last visited april 12, 2005, at `http://www.touchgraph.com`.

Tsintsifas, A. (2002). *A framework for the computer based assessment of diagram based coursework.* Retrieved April 12, 2005, at `http://www.cs.nott.ac.uk/~azt/papers/azt-phd.pdf`. (Dissertation at the University of Nottingham, England)

Veerman, A. L., & Treasure-Jones, T. (1999). Software for problem solving through collaborative argumentation. In P. Coirier & J. Andriessen (Eds.), *Foundations of argumentative text processing* (p. 203-229). Amsterdam, The Netherlands: Amsterdam University Press.

Wang, D., & Zeevat, H. (1998). A syntax-directed approach to picture semantics. In K. Marriott & B. Meyer (Eds.), *Visual language theory* (p. 307-323). Berlin, Germany: Springer.

Wang, J., & Liu, M. (2003). A formal model integration. In J.-P. Tolvanen, J. Gray, & M. Rossi (Eds.), *Computer Science and Information Systems Reports TR-28: Proceedings of the 3rd OOPSLA Workshop on Domain-Specific Modeling* (p. 35-42). Jyväskylä, Finland: University of Jyväskylä Printing House.

Wang, S., Wang, W., & Huang, Q.-M. (2002). Using computers as mindtools to learn time concept in elementary school. In Kinshuk, R. Lewis, K. Akahori, R. Kemp, T. Okamoto, L. Henderson, & C. H. Lee (Eds.), *Proceedings of the 10th International Conference on Computers in Education (ICCE)* (p. 808-812). Los Alamitos, CA (USA): IEEE Press.

Wang, S. P., Yeo, G. K., & Poh, K. L. (1998). An object oriented modelling integration framwork in distribution systems. In J. Gu (Ed.), *Proceedings of the 3rd International Conference on Systems Science and Systems Engineering (ICSSSE)* (p. 75-80). Beijing, China: Scientific and Technical Documents Publishing House.

Wang, W., Haake, J. M., Rubart, J., & Tietze, D. A. (2000). Hypermedia-based support for cooperative learning of process knowledge. *Journal of Network and Computer Applications*, *23*, 357-379.

Weasenforth, D., Biesenbach-Lucas, S., & Meloni, C. (2002). Realizing constructivist objectives through collaborative technologies: Threaded discussions. *Language Learning & Technology*, *6*(3), 55-86.

Wild, M. (1996). Mental models and computer modelling. *Journal of Computer Assisted Learning*, *12*(1), 10-21.

Wilson, B. G. (1996). *Constructivist learning environments: Case studies in instructional design.* Englewood Cliffs, NJ (USA): Educational Technology Publications.

Wilson, R. (2002). *Four colors suffice.* Princeton, NJ (USA): Princeton University Press.

Winter, A., Kullbach, B., & Riediger, V. (2002). An overview of the GXL graph exchange language. In S. Diehl (Ed.), *Software visualization: International seminar, dagstuhl castle, germany, may 20-25, 2001. revised papers.* (p. 324-336). Berlin, Germany: Springer.

Wissenskommunikation project homepage. (n.d.). Last visited april 12, 2005, at `http://www.wissenskommunikation.de`.

Wittenburg, K. B., & Weitzmann, L. M. (1998). Relational grammars: Therory and practice in a visual language interface for process modeling. In K. Marriott & B. Meyer (Eds.), *Visual language theory* (p. 193-217). Berlin, Germany: Springer.

Zhang, J. (1997). The nature of external representations in problem solving. *Cognitive Science*, *21*(2), 179-217.

Zumbach, J., Mühlenbrock, M., Jansen, M., Reimann, P., & Hoppe, H. U. (2002). Multi-dimensional tracking in virtual learning teams. an exploratory study. In G. Stahl (Ed.), *Foundations for a CSCL Community: Proceedings of the 4th International Conference on Computer Support for Collaborative Learning (CSCL)* (p. 650-651). Hillsdale, NJ (USA): Lawrence Erlbaum.