Conceptual Modeling Systems: Active Knowledge Processes in Conceptual Categories

Christopher Landauer[1] and Kirstie L. Bellman[1]

Aerospace Integration Science Center,
The Aerospace Corporation, Mail Stop M6/214,
P. O. Box 92957, Los Angeles, California 90009-2957, USA
{cal,bellman}@aero.org

Abstract. Our application domain is space systems, which are huge systems containing satellites, ground processing and data distribution systems, the launch systems that put the satellites into space, and the manufacturing systems that build and maintain them. Good modeling is essential to making such a huge system reliable and affordable. A Conceptual Model is the first step towards a specification of such a system. It is intended to help the designers invent the system-specific concepts and work out the design decisions and interactions, so that the system design can be evaluated before it is constructed.
We describe an architecture for a Conceptual Modeling system built along these principles, based on our earlier work on Wrappings, which is an integration infrastructure for Constructed Complex Systems, and on Conceptual Categories, which is a very flexible new Knowledge Representation technique, and show how Conceptual Graphs and related tools can fit into the framework.

1 Introduction

Our application domain is space systems [2] [12], which contain satellites, with their on-board sensors and communication subsystems, control segments that are used to manage them remotely, manufacturing systems that build, repair, and replace them, launch systems that loft them into orbit, and usually also contain processing and distribution systems in space and / or on the ground. They are among the most complex systems that we have built, involving hundreds of organizations, thousands of software components, millions of hardware components, billions of dollars, and decades of development time. Traditional knowledge management tools simply cannot cope with this scale of problem, since they are usually oriented towards having individual users.

Good modeling is essential to making such a *Constructed Complex System* effective, reliable, and affordable. Conceptual Models help the designers invent the system-specific concepts and work out the design decisions, so that the system design can be evaluated before the system is constructed. There is an inherent conflict between the messiness of the environments and worlds that we are usually trying to model and the neatness required by our modeling methods, especially when they are defined in particular formal spaces. There is also an inherent tradeoff between the expressive power available in a modeling method and the computational complexity of using it (and sometimes even the decidability of questions about it and its models).

We address both of these problems using multiplicity and context. We describe a Conceptual Modeling system that has many different modeling styles and analytic tools that apply in particular contexts, together with explicit knowledge about when different methods and tools may be applied. It contains a large collection of analysis and display tools and results, technical databases, and previous designs, and it has a system architecture that makes them easy to use appropriately, easy to integrate different information coherently, and easy to modify and evaluate a prospective design.

1.1 Modeling Methods

Conceptual Graphs [22] have been shown to provide a convenient and powerful method for human users to represent certain kinds of declarative knowledge. Several tools have been developed to assist human users in creating the graphs, and there are also some tools that convert between different forms of the graphs. There has been some recent work in including activities in the graphs [19] [4] [14] [15], and in studying constraints on the structure of a Conceptual Graph system [16].

However, Conceptual Graphs, and indeed, any other single modeling mechanism, cannot suffice for modeling a complex system [3] [23], since every model is a simplification that necessarily leaves something out. Moreover, modeling systems based on any formal logic also cannot suffice, since they are limited by the logic they use, in several ways: the normal inconsistencies that occur in early models are either precluded from the outset (limiting flexibility), or they are not apparent and can remain hidden for too long (limiting implementability). Similarly, excess precision interferes with accuracy of modeling, especially in the early design stages, when not enough is known about the implications of assumptions.

Also, most computer programs make design decisions about what will be implemented, and therefore limit what can be implemented. Practicalities of large-scale design also require a range of precisions and formality, so that they do not preclude appropriate imprecision and abstraction. Finally, most personal modeling tools are much too limited in scale for these problems. Imagine, for example, what your favorite tool will do with the tens of millions of concepts and components that go into a space system. These elements must be modeled largely automatically, and with the specific assumptions about their construction all made explicit, so it can be processed.

1.2 Active Knowledge Representation

A complete knowledge representation system should be more than merely a way of recording knowledge in some extractable and analyzable form. It should include the processes that perform all of the manipulations of the representation, as well as the interpreters of all the declarative knowledge (with extensions [19] [14] [15] or not [4]). It is often limitations and ambiguities in these interpreters that render a declarative knowledge representation scheme largely useless; we want to keep these interpreters separate from the representation, so that we can change them without changing the representations (for example, the interpreters of [4], [14], and [15] are different, and we might want to use all of them in a sufficiently complex problem).

If these modeling systems are to grow in scope and applicability, especially in our application area, then several kinds of process expansions are needed: (1) more methods for assisting users in producing and manipulating pictures are needed, (2) more methods for incorporating different representational styles are needed, (3) more methods for structuring the data are needed, especially for scaling up to sizes and complexities that humans are not able to produce or understand individually, (4) more methods for interpreting the data are needed, including displays, analyses, comparisons, conversions between graphical and symbolic forms, etc., and (5) more methods are needed for integration among different formal representations.

In fact, since every model is a simplification, we know that NO one model or modeling method can be sufficient to answer all questions about a complex system [3] [23], so we will need multiple models, multiple modeling languages, and even multiple modeling methods, to address any complex phenomenon. With all of these multiplicities, it follows that we also need integration mechanisms to help us understand whether and how these multiple entities can be fit together, and this leads to our requirement that a Conceptual Modeling system have access to descriptions of all of its own components, so that it can reason about how to apply them.

2 Our Approach

Earlier [11], we described a representational method that is powerful enough and flexible enough that systems built using it can represent, reason about, and change their own structure and behavior. We argued that a purely declarative modeling or programming method does not DO anything, so that something else has to supply the "spark of agency", and that a modeling method that does not specify these interpreters is not complete and is often not well-defined. We showed before that our Conceptual Categories [7] offer a new way to combine many modeling methods, including both declarative and procedural knowledge, into a single coherent framework, and how our Wrapping infrastructure [9] allows us to build Constructed Complex Systems that are themselves knowledge structures, so that they contain all of the interpreters for all of their own declarative knowledge, and that all of those interpreters are explicit and selectable, so the resulting system has no privileged resources at all.

Conceptual Categories for knowledge structuring and Wrappings for system structuring support all of these expansions by (1) separating the representations from the interpretations, so that different interpreters can be chosen in different contexts, and different construction methods can be used; (2) making all of the interpreters explicit, so that they can be compared and analyzed, and new ones more easily integrated; (3) providing a flexible way to define mappings from problems to be addressed to resources that might be applicable in a given context; (4) allowing alternative conceptual representation methods in the same system; and (5) supporting multiple users, cooperating to build the same design.

We showed that this approach provides several advantages for flexible modeling: (1) an explicit place for context and viewpoint in the knowledge structure; (2) an explicit treatment of modeling assumptions; (3) multiplicity of constituent structures according to viewpoint (context-dependent flexibility); (4) multiple interpreters and other procedures for the same declarative knowledge, selected according to context (situated knowledge); and (5) a Computationally Reflective knowledge representation [10].

In this paper, we briefly describe both of these structuring mechanisms, and then describe an architecture for a Conceptual Modeling system, built along these principles.

2.1 Problem Posing

The Problem Posing Interpretation is a uniform declarative style of interpreting all programming and modeling languages [8] [9] It separates posed problems from the resources required to study them. This change of attitude greatly simplifies the remaining discussion, and leads to much flexibility in our Constructed Complex Systems.

Functional, imperative, relational, object-based, and dynamic approaches to programming and modeling, all have a notion of *information service*, with *information service requests* and *information service providers*. In most of these languages nowadays, we connect the two of them by using the same names, but there is no real need for them to be even in the same name space (as long as we have some mechanism for connecting them). We have shown that we can make the connection differently and much more interestingly. Similarly, computer-based modeling languages have notions of structuring services, relation services, and activity or action services for manipulating the structures or elaborating them in time, and the same names are used for the service and for the results of using the service. These languages are defined with a particular "preferred interpretation", which is left implicit in the models.

Once we have made the separation between problems and resources explicit, we can connect them in very flexible and even heterogeneous ways, not only by using the same name (as we generally do now), but we can even include context criteria in mapping (e.g., different resources for the same problem in different contexts). We come back to this point in the next subsection, since it is fundamental to the flexibility of our architecture. The Problem Posing interpretation changes the semantics of programming and modeling languages, not the syntax. It makes a program an organized collection of posed problems, instead of an organized collection of solutions without problems. That difference makes programs easier to understand, because the problems at all levels of detail remain in it.

2.2 Conceptual Categories

Conceptual categories are intended to model our notion of categories of concepts [7] [11]. They generalize sets in four directions [7]: our collective objects have (1) indefinite boundaries; (2) indefinite elements; (3) they allow leakage from context; and (4) the elements have multiplicity of structure.

We have argued these points elsewhere, but the gist of our argument in this paper rests on the last property: multiplicity of structure corresponds to considering the same object or class from different points of view, so that we can use different models of the same phenomena for different purposes. Making viewpoint explicit also allows us to model the modeling decisions explicitly, and to keep track of the modeling simplifications so we can relate them to each other and to the processes that create, change and use them. The fundamental organizational entity is the *category*, which represents anything that can be labeled by a symbol. Categories are descriptions of all phenomena. In fact, we want our notion of category to be co-extensive with the notion of "concept" and the notion of something that we can describe and reason about. Categories work by separating element membership from structure, so that we can define different element structures for different explicit viewpoints. The fundamental organizational relation is the *division*, which is a model of the structure of the elements of the category according to an associated viewpoint. Relationships and actions are also categories, whose divisions are relationships among divisions or programs.

The hierarchical relationships that define categories are simple: a category has *viewpoints* that distinguish *divisions* which contain *roles* for other categories and define relationships among them. The constituent categories participate in the relationship, each with a particular role. Filling a role in a model (identifying an appropriate category) is a fundamental operation, analogous to the use of "surrogates" in [17]. In particular, this means that divisions **are** relationships for categories, and we insist that that ALL structuring devices are in the divisions. This is how we represent our modeling choices, since we can describe what is and is not modeled in the category to get a representation in the viewpoint description.

We use viewpoints to emphasize certain *distinctions* among entities in a category. Distinctions can be differences in content, structure, available information, or intended use. They are often explicit, but sometimes unknown at first. This illustrates a major point about these categories; we can assert information about categories that we cannot compute with the information at hand. This feature is important, because not everything we know about a complex situation can be computed from a description of it.

2.3 Wrapping

In a long series of papers originally aimed at the software and systems engineering community [6] [8] [9] [10], we defined and developed a Computationally Reflective integration infrastructure for Constructed Complex Systems that we called *Wrapping*. It is based on processing explicit qualitative information about all of the system components and their interconnection architecture. These systems are complex collections and interactions of components. They often contain heterogeneous processes, difficult and possibly unknown requirements, and function in complex environments. Designing and building such systems requires explicit models of the system, its architecture, and the environment in which it is expected to operate [21], and suitably flexible computer-based design support [2] [12].

The Wrapping approach to integration has four essential features [6] [8] [9], that underlie its flexibility and power of expression:

1. ALL parts of a system architecture are *resources* that provide some kind of *information service.*
2. ALL activities in the system are *problem study*, (i.e., all activities *apply* a resource to a *posed problem* in a *problem context*).
3. *Wrapping Knowledge Bases* (WKBs) contain *Wrappings*, which are explicit machine-processable descriptions of all of the resources and how they can be applied to problems to support the five *Intelligent User Support* functions [1]:, *Selection* (which resources can be applied to a particular problem), *Assembly* (how to let them work together), *Integration* (when and why they should work together), *Adaptation* (how to adjust them to work on certain kinds of problems), and *Explanation* (why certain resources were or will be used or not used). That is, the Wrappings describe not only "how", but also "why", "when", and "whether" the use of a resource is appropriate for a given problem in a given context.
4. *Problem Managers (PMs)*, including the *Study Managers(SMs)* and the *Coordination Manager (CM)*, are active integration processes that interpret the Wrapping descriptions to collect and select resources to apply to problems.

The first two features provide a kind of consistency of expression at the meta-level to support heterogeneity of content and behavior in the system: the uniformity of treating everything in the system as resources, and the uniformity of treating everything that happens in the system as problem study. The entire system is made Computationally Reflective by treating the PMs as resources themselves: since the programs that process the Wrappings are also resources, and are also Wrapped, all of our integration support processes apply to themselves, too [10]. In particular, the same flexibility of resource use occurs in the system internals, and, in fact, there are NO privileged processes or data sources at all: every part of the system is selectable at run time (provided of course that the resources that do the work have been made available to the system). Therefore, the system has a complete model of its own behavior (to some level of detail), so we can retain design decisions and rationale in the code, and the system can examine and evaluate them. It is this ability of the system to analyze its own behavior that provides some of the power and flexibility of resource use.

Wrapping Processes The heartbeat of the system is in the Coordination Managers (CMs), which repeatedly cycle through a loop of getting a problem posed and using a Study Manager (SM) to study it:

```
Coordination Manager Steps
Find context : determine containing context from user or by invocation
 indefinite loop :
    Pose problem : determine current problem and problem data
    Study problem : use an SM to do something about problem
    Present results : to user
```

There are alternative CMs, including ones that manage the interactions among a number of users, in addition to the simple default CM described in the figure. The default Study Manager (SM) is the main problem solving algorithm. It runs through a default SM step sequence:

```
Default Study Manager Steps
Interpret problem :
    Match resources : get list of candidate resources
    Resolve resources : reduce list via negotiation, make some bindings
    Select resource : choose one resource to apply
    Adapt resource : finish parameter bindings, use defaults
    Advise poser : describe resource and bindings chosen
Apply resource : go do it
Assess results : evaluate
```

The SM process begins with a problem poser, a problem defined by its name and associated data, and the context in which the problem was originally posed. The default SM step sequence represents a basic inline planner.

We get a much more powerful system by making these functions recursive in the “meta-direction” [8]: each step above is also a posed problem, which is interpreted by the same kind of selection process to choose resources to apply to those problems. In each case, there is a simplest resource that applies (for example, the default SM is the simplest resource that applies to the problem "Study Problem"). What we get by using Wrappings is a very flexible way to assign resources to problems, by a kind of "Knowledge-Based Polymorphism" that we have found very useful in our Constructed Complex System designs. It maps problems to resources, from the problem specification in its context to the computational resources that will organize or provide the solution. In particular, Wrappings allow a much more flexible assignment process for matching categories to roles in divisions than would normally be used.

Implementation Considerations The machinery we describe is not too expensive computationally. Flexibility costs, but it does not have to cost very much. Since the processing of Wrappings and the reflective selection of Program Managers is determined by the collection of resources available, we can use a kind of "Partial Evaluation" [8] [9] to take decision processes out of the runtime code, if the results will not change during the execution lifetime of the program. We can even imagine a kind of "Partial Composition" to combine program aspects into programs to analyze and modify the corresponding knowledge structures. We can use the Problem Posing interpretation to select interpreters, taking the declarative structures as problems, with the interpreters as resources. We can also use it for the use of roles in divisions as problems, with categories as resources, and make the basic connection between Conceptual Categories also computed with the same kind of Knowledge-Based Polymorphism.

Such a system is a complex recursive hierarchy of structure and behavior descriptions, contained in our Conceptual Category web, described by Wrappings, and therefore managed by the PMs. Our framework allows almost anything to be used, since we believe that *whatever* method is appropriate to the computational problem should be used, from binary executables to our Wrapping expression notation *wrex*, which is our choice for resources that will frequently need to be customized, since it supports Partial Evaluation [8] as well as our Problem Posing Programming Paradigm [9] (it is our attempt to produce a semantically neutral language).

3 Architecture

In this section, we describe the architecture of our Conceptual Modeling system. We start by describing how it works, and then how that behavior is implemented using the Wrapping processes. In particular, the Coordination Manager provides a basic loop, which poses problems and calls on the Study Managers to address them, after calling on an initial "Find context" resource, as described above. Among these "Pose problem" resources are command line collectors, menu selectors, and graphical input tools. The selected Study Manager gets posed problems and applies selected resources to them, as described above. It uses the Wrapping

Knowledge Bases to determine the mapping from problem to resource, and to find out exactly how to apply the selected resource. Among the generic resources are structure building operations such as Conceptual Graph construction tools, Conceptual Category construction tools, Wrapping Knowledge Base definition tools, etc..

In order to build a conceptual model using this system, a user starts with an application domain (the system initially has one application domain, for domain modeling, and the use of the system gradually builds up a set of them). The overall storage architecture of our Conceptual Modeling system is as follows: the entire system is one Conceptual Category hierarchy, with data structures defined in the divisions and interpreters attached to some of the divisions through the Wrappings. Some of the data structures are for Wrappings, others are for Conceptual Graphs, and others may be defined in other modeling languages (as long as the requisite interpreters have been provided). The interpreters include the Wrapping processes, the Conceptual Graph tools, and programs that can produce, transform, and analyze expressions in whatever modeling languages are used.

We can define categories using any expression notation for the divisions, including any kind of information modeling notations [22], as long as we remember that their use is for each division (because the divisions are the structure-defining entities), and not for the entire category description. Anything that we wish to do with or to a category is a posed problem, and the Wrappings map that into the use of a resource (or a combination of resources) that performs the appropriate operation. So, for example, the initial "Find context" in the Coordination Manager can read in an entire Conceptual Category hierarchy after getting a user selection of which one to use. In order to do this, the initial program needs only the default CM, Study Manager, CM and SM step resources, and the simplest Wrapping Knowledge Base.

3.1 Application Domains

An application domain is distinguished by a collection of assumptions, abbreviations, conventional notations, theories, methods, and tools. They are essentially never completely defined, since using them often leads to further distinctions and new concepts. We hope to have most of the assumptions made explicit, but of course, that is also very hard, since it is hard for anyone to identify their own assumptions.

An application domain modeling language is a domain-specific language for constructing, elaborating, displaying, analyzing, comparing, and evaluating models in a particular domain. Each of these languages contains some basic entities, relationships, and some basic operations and their effects, together with one or more conventional notations for expressing all of these aspects. The mappings from expressions to evaluations go through the Wrappings to find appropriate interpreters.

An application domain contains

- concepts, including definitions and relationships (defined as Conceptual Categories)
- assumptions and requirements (these are identifications and other relationships among the concepts, and constraints on properties of the concepts)
- conventional notations (references to particular notations whose interpreters are resources)
- abbreviations (combinations of concepts that are frequently used together)
- problem contexts (collections of assumptions and assertions defining particular contexts)
- problem spaces (domains in which particular classes of problems can be defined)
- problem specifications for significant classes of problems
- situations (which are partially specified contexts), with motivating applications
- phenomena (these are less well structured items, containing some commonalities and explanations)
- resources, including examples, methods, and tools (one kind of method is to use a tool).
- theories (this includes the usual notion of theories in logic)

The resources in a domain are defined in the Wrappings as resources that address particular problems in particular problem contexts. They need to be supplied with the domain, unless they are already available from other domains, or from the domain-independent utilities.

In addition to these components, an application domain usually has some applications to and from other domains. These are fairly complicated entities that include a domain map and its inverse, both generally expected to be only partially defined, that show when and how to transfer concepts, constructs and resources between the domains, which provides a collection of connections between resources in one domain and resources in the other.

Many of these components are differently significant for different domains, so they may have different definitions or even remain empty. These differences are all different divisions of the corresponding categories. It is therefore important to relate these different domains with their application components or even a notion of simplification and refinement.

Operations on application domains include adding or changing any of these components, mapping an expression or operation from one domain to another, or checking to see if a problem from one domain can be stated in another (so that corresponding resources can be applied). A common operation is the simplification of expressions based on new assumptions or new context conditions, which is an important kind of specialization.

3.2 Structure of System

The Conceptual Modeling system has three parts: the Wrapping infrastructure, the Conceptual Category Knowledge Base, and the application domain definitions.

The Wrapping infrastructure has four layers. At the bottom is the Wrapping Kernel Layer, which contains the default CM, SM, and their default step resources and Wrappings (including the default WKB reader). The next layer is the Infrastructure Layer, which contains other CMs, SMs, steps, WKB readers, an interpreter for our Wrapping expression notation *wrex*, and their Wrappings. These two layers together are called the Wrapping Core. Other layers may define other readers or interpreters for these Wrappings, but the default ones are here.

The next two layers are essentially independent of each other, and together are sometimes called the Resource Layer. The first one is the Utility Layer, which contains all of the domain-independent tools, such as mathematical functions, computational utilities such as graphical display functions, and other language interpreters, as well as their WKBs (and, of course, the readers and interpreters for whatever special notations are used in these Wrappings). The other one is the Application Domain Layer, which contains the resources and WKBs that are specific to the application domains (we might instead consider these to be several different layer entities, one for each different application domain). These Wrapping infrastructure Layers define the execution processes of the system.

The second part of the system is the Conceptual Category Knowledge Base (ccKB), which includes one Conceptual Category hierarchy, containing definitions for several categories at the top, and a collection of resources and the corresponding Wrappings (we call this part of the WKB the ccWKB). For each of these categories, there is a viewpoint "generic constituent", which gives our default structure for each one, as we describe next. In addition, for the category "cat", there is a "generic example" viewpoint, and every one of these categories is in the corresponding division (this is what defines the top of object hierarchy).

For the category "cat", the "generic constituent" division is a collection of pairs "(vp, div)". For the category "vp", the division is a pair "(foc, cxt)". For the category "foc", the division is a tuple "(scope, res, units)", each of which has a division that is just a label for now. For the category "cxt", the division is a set of pairs "(notation, interpreter)", each of which has a division that is just a label for now (the "interpreter" label is assumed to be a problem name).

Finally, for the category "div", the division is a set of alternative structure representation styles, including "product" (a cartesian or direct product that might be partial), "sum" (a direct sum or coproduct that might be partial), "subcat" for subcategories (a special case of direct sum) or "proj" for projections (a special case of direct product), and a few other relationships. All of these structuring relationships refer to roles for other categories, and may contain constraints on those roles. They are matched to categories through part of the WKB devoted to roles as problems (these links are also part of the ccWKB).

Every operation on a category is a posed problem, for which resources are selected through the ccWKB. This is how resources are attached to divisions and filtered through the context in its viewpoint. Among these operations are "read a ccKB", which allows many different specification notations to be used for it. Our default style is a series of "(keyword, value)" pairs (just like the default WKB entries), which is extremely easy to read and interpret.

This list of constituents of a Conceptual Category is one particular division that is useful enough to provide initially. When we add new constituents, we can either use a different division with a different context (when its applicability is limited to a particular context), or we can change the definition of the division, and define what it means when the constituent is empty (for the categories defined earlier using the previous definition). This ability to change the basic ccKB in different ways leads to great flexibility in the representations.

The connection between the first two parts of the system is maintained in the ccKB: "Wrapping" is also a generic example of a category, as are all of the Wrapping components (context criteria, etc.), and the component descriptions we use are particular divisions (with context defined by the particular Layer or application domain, and whatever other information is needed to distinguish the formats of the various subsets of the WKB). Similarly, "problem" and "resource" are also generic examples of a category (i.e., they are in the "generic example" division of the category "cat"), and whatever structuring we define for parts of the space of problem specifications will be a division of one of the categories, as are the various classes of resources.

At this point, we have a system that stores everything in a Conceptual Category hierarchy, interacts with its users using the resources, and transforms problems to resources through the Wrappings, but it has no application domains in which to do any work.

The third part of the system is the application domain, which includes our default definition of what an application domain is and a number of tools that make it easier to define new application domains and change existing ones. All of these resources are in the Application Domain Layer. As we have described before, "application domain" is a category (recall that this means that it is a "generic example" of a category), for which our generic division is a tuple that contains a number of different kinds of entries. These have been described in the previous subsection and will not be repeated here. All of those components are also categories, and have one or more divisions in the hierarchy (different divisions are usually separated according to application domain assumptions).

The initial application domain contains resources for constructing and analyzing application domains, which allows users to build up their own collection of domains. As this Conceptual Modeling system is used, we will build and can make available domain models for various application domains, starting with our original ones of space systems and autonomous agents.

4 Conclusions

We have shown how the Wrapping integration infrastructure and the Conceptual Category knowledge representation mechanism can be used to organize a Conceptual Modeling system so it can use any available tools and methods, and can accommodate any available knowledge bases and other information sources. Since the domain models will contain their context assumptions and requirements, it will be possible to move them to other instances of this Conceptual Modeling system, and maybe even to other Conceptual Modeling systems.

We have started to implement this system architecture, to help us in our own study of the use of ontologies and problem solving methods in Constructed Complex Systems. We are also examining the expressive power of Conceptual Categories, and relating it to their computational complexity, and showing how they relate to the usual categories of Mathematics, so we can have formal underpinnings for our notions of simplification and elaboration.

Many research issues remain, of course, such as what identity and individuality should mean for these systems that will be used in the real world, because they will have to know what the "current situation" is, as opposed to the myriad hypothetical situations that they are prepared to consider, what particular individual items are, as opposed to the classes of items that they are prepared to consider, and who certain individual users are, so that they can manage (for example) whatever authorizations or translations are needed.

Similarly, we need to study how much context is actually needed for medium-size structures, and how the context and other parts of viewpoint are to be expressed (in other words, what are some useful divisions for these categories).

References

1. Kirstie L. Bellman, "An Approach to Integrating and Creating Flexible Software Environments Supporting the Design of Complex Systems", pp. 1101-1105 in *Proc. WSC'91: 1991 Winter Simulation Conf.*, 8-11 December 1991, Phoenix, Arizona (1991)
2. Kirstie L. Bellman, April Gillam, and Christopher Landauer, "Challenges for Conceptual Design Environments: The VEHICLES Experience", *Revue Internationale de CFAO et d'Infographie*, Hermes, Paris (September 1993)
3. Richard Bellman, P. Brock, "On the concepts of a problem and problem-solving", *Amer. Math. Monthly*, Vol. 67, pp. 119-134 (1960)
4. Walling R. Cyre, "Executing Conceptual Graphs", pp. 51-64 in [18]
5. Bernhard Ganter, Guy W. Mineau (eds.), "Conceptual Structures: Logical, Linguistic, and Computational Issues", *Proc. ICCS'00: 8th Int. Conf. Conceptual Structures*, 14-18 August 2000, Darmstadt, Germany, Springer LNAI 1867 (2000)
6. Christopher Landauer, "Wrapping Mathematical Tools", pp. 261-266 in *Proc. 1990 SCS Eastern MultiConf.*, 23-26 April 1990, Nashville, Tennessee, Simulation Series, Vol. 22(3), SCS (1990)

7. Christopher Landauer, "Conceptual Categories as Knowledge Structures", pp. 44-49 in A. M. Meystel (ed.), *Proc. ISAS'97: 1997 Int. Conf. Intelligent Systems and Semiotics*, 22-25 September 1997, NIST, Gaithersburg, Maryland (1997)
8. Christopher Landauer, Kirstie L. Bellman, "Generic Programming, Partial Evaluation, and a New Programming Paradigm", Chapter 8, pp. 108-154 in Gene McGuire (ed.), *Software Process Improvement*, Idea Group Publishing (1999)
9. Christopher Landauer, Kirstie L. Bellman, "Problem Posing Interpretation of Programming Languages", *Proc. HICSS'99: 32nd Hawaii Int. Conf. System Sciences*, 5-8 January 1999, Maui, Hawaii (1999)
10. Christopher Landauer, Kirstie L. Bellman, "Reflective Infrastructure for Autonomous Systems", in *Proc. EMCSR'2000: 15th European Meeting on Cybernetics and Systems Research, Symp. Autonomy Control: Lessons from the Emotional*, 25-28 April 2000, Vienna (April 2000)
11. Christopher Landauer, Kirstie L. Bellman, "Relationships and Actions in Conceptual Categories", pp. 59-72 in G. Stumme (Ed.), *Working with Conceptual Structures - Contributions to ICCS 2000, Auxiliary Proc. ICCS'2000: Int. Conf. Conceptual Structures*, 14-18 August 2000, Darmstadt, Shaker Verlag, Aachen (August 2000)
12. Christopher Landauer, Kirstie L. Bellman, April Gillam, "Software Infrastructure for System Engineering Support", *Proc. AAAI'93 Workshop on Artificial Intelligence for Software Engineering*, 12 July 1993, Washington, D.C. (1993)
13. Dickson Lukose, Harry Delugach, Mary Keeler, Leroy Searle, John Sowa (eds.), *Proc. ICCS'97: 5th Int. Conf. Conceptual Structures*, 3-8 August 1997, Seattle, Washington, Springer LNAI 1257 (1997)
14. Graham A. Mann, "Procedural Renunciation and the Semi-automatic Trap", pp. 319-333 in [18]
15. Guy Mineau, "From Actors to Processes: The Representation of Dynamic Knowledge Using Conceptual Graphs", pp. 65-79 in [18]
16. Guy Mineau, "The Engineering of a CG-Based System: Fundamental Issues", pp. 140-156 in [5]
17. Guy Mineau, "The Extensional Semantics of the Conceptual Graph Formalism", pp. 221-234 in [5]
18. Marie-Laure Mugnier, Michel Chein (eds.), *Proc. ICCS'98: 6th Int. Conf. Conceptual Structures*, 10-12 August 1998, Montpellier, France, Springer LNAI 1453 (1998)
19. Ryszard Raban, Harry S. Delugach, "Animating Conceptual Graphs", pp. 431-445 in [13]
20. Myriam Ribiere, Rose Dieng, "Introduction of Viewpoints in Conceptual Graph Formalism", pp. 168-182 in [13]
21. Mary Shaw, David Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall (1996)
22. John F. Sowa, *Knowledge Representation: Logical, Philosophical, and Computational Foundations*, Brooks/Cole (2000)
23. Donald O. Walter, Kirstie L. Bellman, "Some Issues in Model Integration", pp. 249-254 in *Proc. SCS Eastern MultiConf.*, 23-26 April 1990, Nashville, Tennessee, Simulation Series, Vol. 22, No. 3, SCS (1990)