A System for Visualizing and Analyzing

Participatory Semantics

by

Hung-Chou Tai

Submitted to the Department of

Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for

the Degrees of

Bachelor of Science in Electrical Engineering and Computer Science

and Masters of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 1995

Author_____
Department of Electrical Engineering and Computer Science
May 26, 1995

Certified by_____
Carl Hewitt
Thesis Supervisor

Accepted by_____
F.R. Morgenthaler
Chairman, Department Committee on Graduate Theses

A System for Visualizing and Analyzing

Participatory Semantics

by

Hung-Chou Tai

Submitted to the Department of

Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for

the Degrees of

Bachelor of Science in Electrical Engineering and Computer Science

and Masters of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 1995

ABSTRACT

Participatory semantics is a method used to describe the semantics of participation in activities. Visualizing and analyzing participatory semantics is very useful in understanding this semantic technology. This system allows a user to create and modify diagrams which utilize participatory semantics constructs and methods. It will check for inconsistencies in semantics and also be able to deduce additional knowledge using derivational calculus with rules from mathematics and logics. Written in highly portable C++, the system will aid in the development of the Continually Available Telecomputing Services Infrastructure (CATSI) with its near-term goals being the TeleVisit project for the Museum of Science in Boston.

Thesis Supervisor: Carl Hewitt
Title: Associate Professor, Electrical Engineering and Computer Science

Table of Contents

List of Figures

Chapter 2

Chapter 3

Chapter 4

Chapter 5

Chapter 6

Chapter 1 Introduction

1.1 Goals

A system for Visualization of Participatory Semantics (VOPS) has been built. Participatory semantics is discussed in [Hewitt & Manning 94] as formal methods for the semantics of participation in activities. Semantics and participation are interrelated and self-reinforcing. According to [Hewitt & Manning 94], participation includes representation, i.e., "no formal methods without participation"; and using formal methods informs future participation, i.e., "accountability is the norm." The formal methods and rules of participatory semantics thus are used to model the present and to use the information that has been gathered to derive and inform any future activity. An activity can thus be anything such as a meeting, a visit to a museum, or a research project. Participatory semantics will help to model the meeting, the trip, and the project, and it will be used to influence the people who have participated in those activities.

Creation of VOPS is geared towards the short-term goal of developing a system for a place such as a museum of science. Called TeleVisit, this system will incorporate the Continually Available Telecomputing Infrastructure (CATSI) outlined in [Hewitt & Manning 94]. [Hewitt & Manning 94] talks about CATSI as a new paradigm for computing which will incorporate wireless interconnect to allow users to continually negotiate events and create semantics to represent the activities in which they are involved. CATSI can become many things to many people. It can be used as a project manager for a corporation, as a personal secretary that can screen messages, or even as a scheduler for a visits to a museum of science. Museums of Science are fertile testing grounds for a near-term implementation of CATSI. A system, called TeleVisit, is in the works as a precursor to larger-scale implementations of CATSI. For the rest of this thesis, examples will be drawn within the context of a museum of science.

TeleVisit resides within a museum as a system which will not only serve as an interactive exhibit for visitors to the museum, but it will also be a mobile device for supervisors to use to lead around groups of people. A supervisor who chooses to use this tool will carry around a mobile computing platform that will allow him to communicate with others who also have such a device and also to query the system itself. The system will include a central computer which will contain information about all the events taking place

within the museum.

Each mobile computing platform and the central computer in TeleVisit will contain databases for storing information. The central computer will hold a database which will be a repository for multimedia clips, sound files, and other information about the exhibits and events in the Museum of Science. It is updated daily with new exhibits and schedules. The databases on the mobile platforms will be used to store information regarding the person's movements throughout the course of his visit. TeleVisit will model knowledge gained from a visit using participatory semantics and be able derive further knowledge using the semantics technology.

Using participatory semantics, for example, TeleVisit can be used to solve problems that may arise during a visit. Consider a teacher leading a group of students around the museum. The teacher can communicate with another teacher who is leading a different group. If a child is lost, the two can communicate with each other using TeleVisit or even ask TeleVisit itself to determine where the child was seen last and when he/she was seen. The lost child's current location can then be narrowed down and the task for finding a lost child is simplified. The task of finding a lost child will be described in **Chapter 3** as one of the features a CATSI system like TeleVisit can incorporate.

In addition, participatory semantics can be used to schedule events. A supervisor can just input the exhibits that he wishes to see and the demonstrations and shows he would like to attend, and TeleVisit can present a suitable schedule. Finally, tickets can be bought, and questions can be asked about exhibits resulting in a much more interactive and participatory trip to the museum. At the end of the visit, each user can be given a printout of their daily activities, and a list of further activities (further readings, exhibits that were missed, etc.) that the user may wish attend to later. This printout will be one way in which TeleVisit influences further participation.

TeleVisit will help automate, negotiate, and annotate much of the activities that occur within a visit to the Museum of Science. Those tasks are similar to many of the tasks that people need to perform at work or while planning their own activities; thus, TeleVisit will help pave the path to a possible implementation of CATSI on a larger and more complex scale. Eventually, CATSI will be implemented on an environment with a heterogeneous user group (i.e., the user group will not be restricted to computer literates and scien-

tists) in a massively-networked environment such as the Internet. Participation in CATSI will just require computers to recognize the CATSI protocols, and access can immediately be gained.

1.2 Technologies Required

Many technologies are needed for successful implementation of TeleVisit and CATSI. One technology that is required is the use of wireless communications. To fulfill that need, wireless ethernet links with radio or infrared communications can be used. For computing to be continually available, users should not have to find a place to plug their computers into a network. Wireless ethernet PCMCIA adapters are available which can be plugged into a portable workstation. Access points can be set up between locations to broadcast and receive packets. Also, cellular communications can be used to fulfill the need for wireless communications. It allows connections between computers even when there are no networks in sight. A modem can be added to a portable computer and attached to a cellular telephone to allow a user to access CATSI using services from a cellular phone company.

Another technology being used for the deployment of CATSI is the use of objects and object-oriented technology. Persistent object bases will be employed in each of the mobile workstations. This new technology means that data will be stored as objects in the database. Presently, relational databases are being used to store large amounts of information. Relational databases store information in tables. Object-oriented databases use persistent objects which can reside in two types of places--permanent storage such as hard disk or CD-ROM and temporary memory such as RAM. Persistence allows for objects which reside in permanent storage to be stored in almost the same way as in temporary memory, eliminating the need for wrapping and allowing for faster access between objects.

The persistent object base used in CATSI will incorporate a distributed objects paradigm in which the database management system manages vast numbers of objects over an entire network. Each machine on the network will contain its own object database which other machines can query. Thus, every machine can be viewed as both a client and a server, having the capabilities to request objects from other machines and grant access to objects from its own object base to others [Orfali et al 95]. Using a persistent object base in a distributed object paradigm is a suitable mean for implementing CATSI as users of these CATSI systems

will create their own information and share their data objects with other people on the network. In TeleVisit, users will constantly be creating semantic objects which represent the activities and objects which they are related to and sharing those objects with other visitors within the museum. The objects they create will constantly be modified by themselves and even by other people so a good object base and object base manager will be required.

The technologies that are currently needed to implement CATSI are what can be described as "bleeding-edge". They are still in their infancies, but they promise to provide the usefulness and functionalities which CATSI needs. Upon the maturity of these tools, many of their bugs and limitations will have been eliminated and TeleVisit will be a viable tool for introduction to the public. Another fundamental technology, however, is needed. It resides within the realm of artificial intelligence and is required in order to deal with the more important issues of semantics and negotiation within CATSI. This technology is called participatory semantics, and the remainder of this thesis will discuss the Visualization of Participatory Semantics (VOPS) system that has been created.

1.3 Outline for the Remaining Chapters

Thus, the remainder of this paper will consist of five chapters.

Chapter II will outline the goals of CATSI, and how TeleVisit will be used to test how those goals can be achieved. CATSI will incorporate two interrelated ideas. The first one is that the system has to be a mediation tool for the interpreting meanings of the inputs that a user enters. CATSI will have to disseminate information and be able to filter out their basic meanings using formal methods that have been built into CATSI. As more users enter knowledge into the persistent object base, determining correctness of conflicting information and necessity of data will create complexity and increase computation time. The second closely-knit idea is that the system should be able to negotiate when problems between users occur. Negotiation will arise if mediation fails to generate the desired or correct response. Finally, Chapter II will also discuss some of the issues of multi-user knowledge base systems. These issues will come up later in the design, implementation, and usage of CATSI and TeleVisit.

Chapter III will give details on the participatory semantics technology that will be used in CATSI

and is also the basis for VOPS. Participatory semantics will be used to address some of the goals and requirements of CATSI. The necessary condition that CATSI be a survivable system and also the requirements for survivability will also be discussed. Since participatory semantics seeks to be the foundation upon which CATSI is built, it has to incorporate many of the survivability features that CATSI requires. The terminology that will be used will be introduced, and key constructs such as consistency checking and knowledge derivation will also be discussed. In addition, the methods and rules that have been constructed for use in VOPS will be presented, and examples will be given to illustrate those semantics.

Chapter IV will discuss the VOPS system that was built. Technologies used to implement the system will be discussed and the reasonings behind their use will also be given. VOPS is designed and implemented in the Windows environment, and it uses Microsoft Visual C++ and Viewsoft's Utah. These two technologies will allow extensibility to the TeleVisit project and also any further implementations of CATSI. Furthermore, Chapter IV will talk about the programming methodologies used. The class hierarchy used in this system will be given. Included with the class hierarchy will also be a description of each subclass created along with all of its respective interfaces. Pseudo-code which describes the algorithms for some of the derivational calculus and consistency checking routines will also be presented and explained. At the end of this chapter, the reader should have a very good understanding of how the VOPS system works and the backbone behind why it works.

Chapter V contains examples of scenarios that may occur when semantics are entered into VOPS and how the system handles some of the conflicts and inconsistencies which may appear while creating the semantics. Examples are used to demonstrate the functionality of VOPS and the feasibility of the algorithms discussed in Chapter IV. These examples will also correlate to some of the fundamental rules and inference mechanisms outlined in Chapter III. Sample screen shots will also be shown as examples are stepped through and described.

Finally, Chapter VI will wrap up the discussion on participatory semantics and its applicability to TeleVisit and CATSI. Further extensions and enhancements to the VOPS system will be given. In addition, areas in which more research and better algorithmic design can be done will also be discussed. Finally, limitations and restrictions that this technology may place on resources required will be given.

At the end of this paper, there should be a clearer understanding of participatory semantics and how VOPS can help visualize and analyze scenarios that may occur when using this semantics technology to model activities of participants. In addition, VOPS will help in the further development of the artificial intelligence needed in the implementation of CATSI and TeleVisit. The Museum of Science will provide a good basis for future development and extensions to VOPS and provide a good framework within which to test some of the fundamental concepts of CATSI. With the knowledge garnered from VOPS, future work can be done to extend and enhance participatory semantics.

Chapter 2 Issues in CATSI

One of the main goal of CATSI is to create a system which will use participatory semantics to allow participants (or users) of a system such as TeleVisit to model their activities and find and solve problems that may arise. Solving problems involves two interrelated ideas, mediation and negotiation. Mediation of meaning will be used to interpret the meaning of the information that the user enters into the system, while negotiation will be used when problems arise with the information that the user or users have entered. The two ideas are coupled in that negotiation is required when the mediation breaks down and sometimes negotiation is needed in order to facilitate mediation. These two concepts will be further elaborated in the first part of this chapter.

CATSI will use participatory semantics which incorporates multi-user knowledge representation technology that will model the activities which take place among participants. So the latter part of this chapter will discuss the issues concerning a multi-user knowledge representation system. Problems in a system such as TeleVisit will involve the heterogeneous user base that will be developing and using it as a tool and also the different models of the world that users will create and put into TeleVisit.

2.1 Mediation and Negotiation

Two concepts which TeleVisit and CATSI have to incorporate into their modelling methods and semantics are the ideas of mediation of negotiation. In systems which involve CATSI, users will constantly be making representations of the world based on their perceptions of the environment around them. These representations will constantly be evolving as users gain better understandings of the world.

Analogies can be drawn to industry. Take the structure of a company, for example. In the early years of big corporations, companies were very hierarchical with many layers of management, and climbing up the corporate ladder to gain prestige and responsibility required patience and seniority. Now, however, companies have become flatter and less management-oriented so that projects can be orchestrated more smoothly. The mobility created allows people to gain prestige without moving up the corporate ladder. As the skill set and demands of workers change so too does the infrastructure of a company. A company's per-

ception of what is required in the real world has been evolving in the past few years to incorporate that new company structure. The same analogy can also be applied to people's representations and perceptions of the world around them.

Just like a company's organization changes, so too do people's perceptions. And just like many companies have different organizational structures, so too do people all have different perceptions of the same environment. Although those views of the world around them are all different, they may all be correct. And although they may be correct, they may yet still be inconsistent with each other. The idea of consistency is not an issue to be discussed in this chapter. The more important task, however, is the mediation between all the different ideas which different people may have. These ideas are all the result of the different activities in which people participate and how those activities have influenced the people's actions.

Mediation involves interpretation and finding the right meaning of a representation which a user or users input. In CATSI, every person will receive a portable computing platform which will contains its own persistent object base to allow users to create their own representations of their activities and relate those activities to other users' activities. Many people will be creating different semantic representations of the world. These will all become messages which are sent to the system, and they will have to be dealt with in some way. Some messages can be dealt with by the system, while others may be sent back out to different participants who are using CATSI. This model of the mediation system can be seen in **Figure 2-1**.

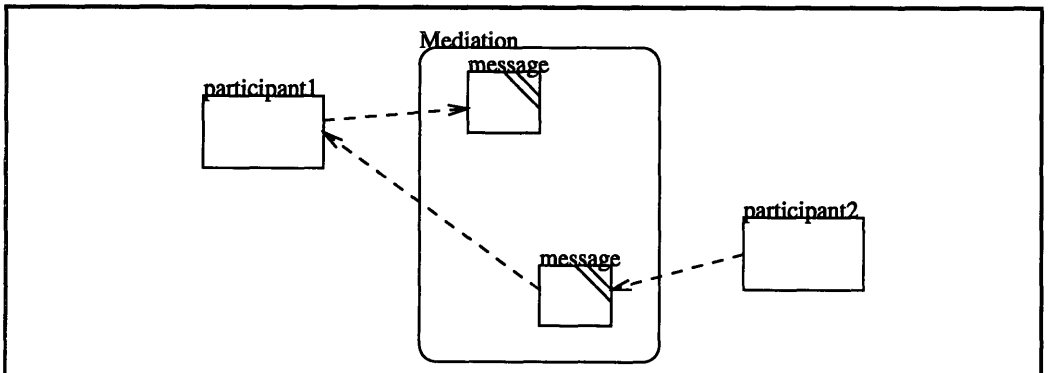

Figure 2-1: Mediation architecture model to be used in CATSI.

Mediation will be used to deal with these messages, routing them to their intended participants and processing them as needed. "As needed" means that some messages do not require CATSI to perform any compu-

tation but does require input from a user to determine further action.

Upon mediation, the CATSI can inform a user of impending events or even derive more information that the user may find useful. Mediation services provided by TeleVisit will thus be a way to organize,. say, a trip to the Museum of Science. At the beginning of a visit to the museum, supervisors of groups are given portable computing devices which are connected wirelessly to a network. The supervisors will schedule events for the day and communicate with each other when problems or contingencies arise. All of the requests that the supervisors demand from the system will result in messages to the mediation architecture, which will interpret the messages and perform the necessary computations.

When mediation is not able to provide the appropriate meaning, negotiation is needed to deal with that contingency. Negotiation is the means whereby two participants communicate with each other to resolve any problems that may have arisen because mediation did not work. Take an example that may arise in the Museum of Science. Two supervisors from the same school wish to coordinate their visit, and each requests to see a particular show. If mediation works, then they will both be able to see both shows. Suppose, however, the shows overlap and there are no more shows for the day. Then negotiation will start. Supervisors will tell TeleVisit perhaps their interests and what other exhibits or shows they may want to see for the day. Based on the new information, the negotiation architecture will find a solution which is satisfactory to both users. In the process, it may query both supervisors for more information or it may just use whatever it already has in its knowledge base.

These two concepts or mediation and negotiation will be the keys to a successful deployment of CATSI. Using participatory semantics, a technology which will be discussed in the following chapters, parts of these concepts will also be implemented in VOPS.

2.2 Multi-user Knowledge Representation Technologies

The goal of CATSI is to ultimately create a system in which "telecomputing systems can be used effectively at times and places where they are not currently available" [Hewitt & Manning 94]. In particular, CATSI serves to use negotiation to influence future participation. This concept will be discussed in detail in the next chapter as it pertains to the participatory semantics technology which will be used in the implemen-

tation of CATSI. Mediation, which was also discussed earlier, is the other underlying concept in CATSI and will always be in use whenever the user performs an action. One of the key concepts, however, regarding negotiations is the fact that many people will be using this system, putting their own views of their activities into it, and requesting the participatory semantics technology to sift through and sort it. Because many participants will be in the system requesting different services, problems will arise. First, participants in CATSI will be explained. A participant is a broad term referring to any type of object, formal or informal. Formal objects represent objects that reside in the real world. These can range from physical objects such as people and things to other types of objects such as meetings and schedules. Formal objects will be called episodes in participatory semantics. Informal objects are more conceptual than real and refer more to the relationships between objects. These informal objects will be called links in participatory semantics. Examples of links are "contains" and "before" links, as in "the group contains a supervisor" and "the electricity show is before the reptile demonstration." All of these participants, whether formal or informal, will be objects that reside in different machines distributed throughout the network.

Many of the problems that CATSI systems will have to deal with stem from the fact that CATSI will almost always be multi-user. Most existing knowledge representation technologies reside on single user machines or on systems with a homogeneous set of users [Basu 93]. These are users which all have a general sense of how to use the knowledge base effectively and will all input information into the knowledge base the same way. These knowledge bases are thus termed "monolithic" [Meseguer 93]. In such a system, there is a single set of rules which the system searches every time it is given new information. Problems arise when different representations need to be integrated and understood by all the systems in a software system. With CATSI, systems such as TeleVIsit will have to allow different rules to be developed for different users of the system since the user base will always be heterogeneous, and they will all have different interpretations and representations of their activities.

With different participants in different systems, each with different views and representations of the world, the complexity of the entire knowledge base can slow down the processing of a multi-user representation system. Many issues are involved with which CATSI has to take into account before it can hope to present a suitable representation technology. Some issues have already been addressed in the realm of sys-

tem architecture integration. These issues will also be discussed within the scope of knowledge representation.

[Basu 93] has brought to attention many issues that will arise in a multi-user knowledge-based system (MKBS). First, an MKBS requires that appropriate inference and control mechanisms be developed. One problem with any control mechanism on an MKBS will arise regarding whether a specific inference will be restricted to one set of values or multiple sets. Should one set of rules or multiple sets of rules apply to derive information? If the former control mechanism were chosen (one set of rules), then users will be restricted to only one perspective when the system is solving a problem or trying to derive further information from the knowledge base. However, if the latter case were chosen (multiple sets of rules), the system will have to keep track of all the possible values which may arise when it is solving a problem. Since many of the rules are sensitive to the context from which they were derived, some of the desired outcomes may not make sense to any of the users of the knowledge base. Furthermore, if there is a need to backtrack or roll back a problem solving process, there has to be a way of maintaining control of that rollback [Basu 93]. These two results will require substantial processing and storage requirements and may not be feasible for a real-world system.

Another control problem arises when only one knowledge base, instead of the whole network of knowledge bases, needs to be consulted when deriving the solution to a problem. Although most processes in CATSI will involve participants spread throughout the system for any kind of problem solving or negotiation, the performance of single-user problems will be greatly reduced. A mechanism needs to be created that would allow inference rules to store the location from which they were derived [Basu 93]. One easy mechanism used by [Basu 93] is the use of indexes for each knowledge base. These control indices are prefixed onto the inference rules so that only rules that have the specific knowledge base index can be used when multiple sets of rules are available.

The third problem that arises in the control of a multi-user knowledge-based system concerns the rule translation process. If this process has to traverse multiple knowledge bases then there is no guarantee that any final or intermediate result will make sense because the set of rules that the translation process uses may not be the set desired. The rules used could be a cornucopia of rules from many different knowledge

bases. When translating a rule from one knowledge base to another, some mechanism is required to determine which knowledge base should assume control of the problem. One solution that has been presented is that all of the systems may attempt to solve the problem and a mechanism at the end of the process will sort out the results. Yet another solution suggests building a system on top of the representation systems with intermediate rules to route the problem to appropriate knowledge bases [Basu 93]. Both of these solutions to the problems in rule translation however may prove expensive in terms of processing time and extra resources required and may not be worth implementing.

Finally, there is always the problem with inconsistencies between knowledge bases. There are many problems of maintaining or even checking for inconsistencies across different knowledge bases on a knowledge-based system. The extra overhead involved and resources required may be too costly. Then, of course, there are the same problems with rectifying those inconsistencies. With those prevailing problems of efficiency and overhead, one could also ask these questions: Who is right? Who is wrong? Do we care? But even those questions may be irrelevant since everyone may be right based on the information they were given or the assumptions they have made. Dealing with those questions may again require extra processing power and bandwidth, both of which may be scarce. VOPS will attempt to address the inconsistency problem in ways which will minimize overhead by only involving those participants that are related (connected) to the source of the inconsistency.

The problems described above for multi-user knowledge representation systems are very similar to the problems right now in the client/server environment. Parallels can be drawn between the requirements needed in artificial intelligence and those needed in software systems. For example, there is a plethora of database systems and GUI front-ends, and there are development tools that seek to combine all those into usable packages. If all of those components are thought of as participants in a multi-user knowledge base described above, then solutions used in the software systems arena may be applied to the knowledge representation systems arena.

Many solutions in software systems have involved "wrapping" components of systems with common code and interfaces [Landauer & Bellman 94]. Wrapping involves adding an additional software layer on top of another exclusive and proprietary piece of software to provide a common interface for disparate

software systems. For knowledge representation technology, wrapping can have its benefits. It makes everything uniform in terms of description, process, and the methods for describing and selecting processes. Wrapping knowledge bases can also encapsulate information regarding when a knowledge base or a rule within a knowledge base is useful for a particular problem and the requirements for its use. Additional information regarding the assumptions, limitations, scope, and styles of use may also be wrapped to give more information to other systems [Bellman 94]. [Bellman 94] espouses the wrapping approach because with ever-changing and newly-developing software systems, writing wrappers on top of components will allow systems to be more open and usable by other systems.

CATSI thus seeks to use a suitable knowledge representation technology as its foundations for allowing users to create and represent interdependent activities. The issues that it seeks to deal with are high-level ones that are analogical to human reasoning and interactions. These issues involve mediation to attempt to understand the meanings behind different user representations and negotiation to try to settle problems that mediation cannot handle. Other issues that CATSI faces are problems with control that are similar to those that confront all multi-user and multi-model systems. Control and inference mechanisms all face problems which multiply with additional knowledge bases or users. The wrapping approach used in large-scale software systems seems to be able to alleviate many of the problems prevalent in software systems and can be adapted to knowledge representation systems. CATSI will use the idea of participatory semantics as a foundation for dealing with these issues. This representation scheme will be discussed in the next chapter.

Chapter 3 Participatory Semantics

3.1 Survivable Systems

"CATSI is a fundamental paradigm shift in the ways that people interact using telecomputing technology" [Hewitt & Manning 94]. Participatory semantics will be the technology used to address the fundamental issues of CATSI. Participatory semantics is a mathematical study of formal methods. Some of the issues that had been discussed have dealt with problems of multi-user knowledge-based systems, problems in switching from monolithic to multi-model systems, and the need to establish suitable frameworks which will deal with these problems. Using mathematics and logical inference, participatory semantic technology will help to resolve those issues and become the foundation of survivable software systems such as CATSI and TeleVisit.

A suitable framework will be required in order to present an accurate semantics of the world which is to be modelled. A "survivable system" is the framework in which the participatory semantics model will follow. A survivable system means "having the qualities needed for a system to survive in, and to interact with an environment that is not under its control" [Talcott 94]. Such systems will gain robustness through an ability to represent its behavior, deduce more knowledge about itself, and effectively change its semantics of its own behavior. These ideals will require a firm mathematical foundations in order to build the necessary software system.

According to [Talcott 94], a survivable system will need to interact within an environment that is not under its control and to adapt to any changes in its environment. Participatory semantics will be the basis under which TeleVisit operates. Users of TeleVisit will be changing and evolving their environment using participatory semantics and even changing the environment under which participatory semantics operates. As a result, the system will have to adapt to different user semantics of the world. In the Museum of Science, for example, different supervisors will want to schedule their activities in different ways. TeleVisit will constantly be updating locations of each group of students and determining if conflicts may arise because of their changing locations.

Others have also advocated the use of adaptable systems. These systems will be self-adaptive and

have a fine-grained internal structure that can be tuned dynamically to the environment through interactions and feedback. Systems will have to exploit the world, using it as a resource. They will maintain multiple models internally of the external world and be able to constantly redefine and reconfigure themselves dynamically [Bobrow & Saraswat 94]. Being a totally open system will be the goal of CATSI which will allow it to survive into the future. Users will be the ones ever-changing and ever-evolving their own models of the world. CATSI systems will be adapting and reshaping themselves continuously in order to satisfy user requests and constraints.

Survivable systems also have to robust. It should be able to survive the rigors of a heterogeneous user base which may or may not fully understand the semantics of the technology. The skill levels of the user base may range from developers who are implementing CATSI systems to children who are just using it as a tool to aid in their museum visit. CATSI should be able to tolerate all different kinds of skill levels. For the Museum of Science, that robustness will be partially achieved through TeleVisit. TeleVisit will be the application layer that will be added on top of the participatory semantics foundation. It will have higher level objects which correspond to museum objects such as exhibits, shows, and theaters. These common icons and interfaces will simplify much of the processing logic that will have to be added to the participatory semantics implementation, reducing many unnecessary checks for inconsistencies and conflicts which may arise later. TeleVisit will create all the necessary semantics for the persistent object base based on the queries and input that the user makes.

Finally, a survivable system has to be reactive. Being reactive means that the system will be able to react to changes in the environment. Participatory semantics has its foundations deeply rooted in artificial intelligence and knowledge representation in order to derive additional knowledge from what is already present in the environment. Participatory semantics will be using rules and other constructs such as deductive inferencing schemes to constantly update the environment. Moreover, it will be constantly creating and updating new relationships between participants in the system.

Finally, to achieve survivability, participatory semantics will be using mathematics to study formal methods in knowledge representation. Mathematics is "timeless" [Bellman 94]. It is unfettered by its environment and its constructs and semantics always hold their meanings regardless of context. For example, an

"=" sign has remained unchanged in millennia of mathematics; wherever it may be used, it always takes on the same meaning. Other symbols stemming from mathematics and logic include "if-then" statements and boolean operators. In addition, mathematics has the advantages of being a precise and formal language. In order to describe scenarios in the real-world and to perform deductive inference on the information, the precision and formality which mathematics offers is a necessary foundation in the building of a survivable system. Mathematics thus is well-suited to be the foundation upon which participatory semantics is built. Many properties from mathematics such as transitivity and reflexivity will also be used in participatory semantics for knowledge derivation, and much of the same logic used in mathematics will also be used for consistency checking.

Robustness, adaptability to the environment, and reactivity are thus characteristics of survivable systems. Participatory semantics will require all those traits in order for it to be a good foundation for satisfying the goals of CATSI. With mathematics as a framework for constructing participatory semantics, CATSI should be able to handle most issues which may surface.

3.2 Relevant Issues in Participatory Semantics

There are three issues that a multi-user continually available telecomputing system must resolve; inconsistencies, conflicts, and timing constraints are all problems with which participants will have to create semantics, mediate, and negotiate in order to discover any avenues for resolution and influence how interdependent tasks can be performed. With mediation by the system, participants are free from further involvement. But with more complicated problems, participants in the negotiation process will have to create semantics for past, present, and future participation and use negotiation to reach a solution [Hewitt & Manning 94]. These additional semantics may include tasks to be performed, products to be consumed, created, or compared, resources to be used, and the dependencies between all of the participants. Within the scope of TeleVisit, these semantics describe the types of exhibits which will be seen, the multimedia presentations which the user may request, the amount of computing power that is needed, and how the users can interact with and relate to the exhibits.

3.2.1 Inconsistencies

One of the issues which participatory semantics hopes to deal with is the problem of inconsistency. An inconsistency results when information is given that contradict another piece of information but does not preclude any events from taking place. Inconsistencies can arise in many different scenarios. They may arise when different users create different representations about a scenario. For example, suppose userA thinks that *exhibit1* is in *3 West* of the museum, then she would have this lexical representation inside TeleVisit:

***contains** (3 West, exhibit1)*

UserB may say that the *exhibit1* is not contained in *3 West* but is contained in *2 East*, so he would add these additional representations:

***~contains** (3 West, exhibit1)*
***contains** (2 East, exhibit1)*

All of these representations are legal and they may or may not be correct. However, the main issue here is that when taken together they create an inconsistency. At first, mediation can be used to resolve this example. For example, if coordinates for different location objects have been stored into the database, TeleVisit can check that the physical coordinates of *exhibit1* are within the bounds of a certain location. If no provision has been made to store location information, negotiation will come into play. The system will query all participants involved (*userA*, *userB*, *exhibit1*, *3 West, and 2 East*) to gain more information. In any case, three things may happen. UserA is correct and *exhibit1* is in *3 West*; userB's first statement is correct but not the second which implies that the exhibit is not in *3 West* nor in *2 East*; finally, both of userB's statements are correct and the exhibit is in *2 East*. Negotiation will allow TeleVisit to choose the "correct" representation based on what the participants have to say.

Inconsistencies may also arise when only one user presents contradictory information about a scenario. For example, suppose userA makes this representation

***contains** (group1, person4)*

and then later creates another representation

***contains** (group2, person4)*

To which group does *person4* belong? She might have belonged to both groups at one time or another. Or, userA might have made a mistake thinking that in the second scenario another person, *person2*, is actually *person4*. There is no way that any semantic system can rule out any of the statements in this case. The concept that a person is contained within a group was defined by the user, not by any natural or systematic means. I.e., the user explicitly placed *person4* into a specific group. Therefore, no technical wizardry such as the bounds checking used in the first example can be done.

These two examples show the problems which participatory semantic may have to resolve or even choose not to resolve. Sometimes the inconsistencies are easily taken care of by checking physical bounds as in the first example. But an example like the second one cannot be solved through deductive inference on any system.

Furthermore, imagine having to derive additional knowledge from the second example. Which representation should the system use? Participatory semantic technology seeks to use negotiation to resolve the issue. For the second example, the participant(s) involved may be prompted of the inconsistency and be asked to resolve it. This will simplify any consistency checking that may be required in the future. Or, as in the first example, the mediation used in CATSI can try to take care of the problem first without querying the participants. Whichever is the case, taking care of inconsistencies early on will cause fewer complications in the future.

3.2.2 Conflicts

Another issue that participatory semantic seeks to address is conflicts. Conflicts arise when activities interact in such a way that prevents all of them from being completed. [Hewitt and Manning 94] suggests that conflicts arise when there are both local and interdependent activities. A local activity only involves the system at hand, while interdependent activities require participation of other systems. Since participatory semantic will be the basis for a multi-user, multi-model system for TeleVisit, this will inherently imply that both local and interdependent activities will be present. The interactions which take place will inevitably cause conflicts.

One of the interactions between local and interdependent activities is late-arriving information

from other ongoing activities. A process that is in its advanced stages of processing may find that information crucial to its decision-making process has arrived. The representation system has to make critical choices to deal with that contingency. It can choose to ignore the new information, finish processing the original activity and then reprocess the activity with the new information, or it may stop everything it is doing and redo the processing, taking into account the newly-arrived information. A decision has to be made somewhere to take a specific course of action.

Another interaction is with multiple local authorities. Local authorities allow participants to react immediately to changes in the environment. These agents can deal with conflicts as soon as they arise. However, conflicts may arise when multiple local authorities all compete for local resources. Since each authority acts locally with its own local information, it may not know of another authority who might be dealing with a similar request which demands similar resources from the system. More negotiation might have to be done to resolve these resource-sharing issues.

Finally, arm's length relationships can also cause conflicts in systems with both local and interdependent activities. Local participants often act with arm's length relationships with other participants. These relationships are used when participants do not care how the other non-local participants react to arriving information. Encapsulation of each local process ensues and shelters other activities from the inner workings of other processes. Encapsulation is beneficial because it hides the complexities of one process from those of another, but it can also be detrimental when local activities begin to develop entrenched and incompatible activities which others may not be aware.

3.2.3 Timing and Interruptions

The last issue which participatory semantic wishes to deal with is the timing and interruption problem. With continually available systems, users may be inundated with information, most of which useless and uninteresting. This same problem is prevalent today as people are constantly receiving unwanted, junk mail from soliciting vendors. CATSI will alleviate that problem by prioritizing messages, filtering out unimportant ones, and more or less acting as the user's personal secretary. All of these things will be done using the structure of already-established representational activities which people use all of the time. These

include day planners, address books, and project management tools. Participatory semantic will be able to signal when important events arise, help people plan out their activities, and finally summarize the day's activities, which are all extremely relevant in the TeleVisit system. In the Museum of Science, participatory semantic technology will notify users of interesting shows and demonstrations, help plan their trip around special events at the museum, and finally give users printouts of their day. Recommendations on any other reference materials that are relevant to the user's interests can be made based on the types of exhibits that the person saw.

Inconsistencies, conflicts, and timing and interruptions are three of the main issues that participatory semantic hopes to deal with as it seeks to build a suitable semantic foundation for CATSI. Before undertaking a discussion of participatory semantics, the basic terminology that will be used in ensuing chapters have to be described.

3.3 Terminology

Participatory semantic is a technology in which participants create representations in order to perform tasks, negotiate about conflicts, and influence other activities. Thus, the underlying principle is a description of the world around it. This description will contain objects (the participants) and all the relationships between the objects.

The term which will be used to describe any type of object in the world is an "episode." An episode is a broad term which will describe anything in the world. The "thing" in ***anything*** can be an object such as a person or a book, an idea such as a thought or a phrase, or even a process such as an event or the schedule of the event. Anything which exists in the world can be an episode.

Another term which will be used for the rest of this paper is a "subepisode." A subepisode is any episode which resides within another episode. So, an episode can be thought of as a container for other episodes and also other subepisodes. One important idea about subepisodes in the present implementation of VOPS is that they cannot stand alone. They are always created within an episode and destroyed within the episode in which they were created. How subepisodes actually fit into the participatory semantics scheme will be described in more detail later in Chapter IV.

Finally, there is the concept of "links." Participatory semantic not only involves objects and participants, but it also deals with the relationships between all of them. These relationships will be made through the use of links. Links will be the connections between different episodes. A link will thus be an informal object in participatory semantics, informal in the sense that it will not be any object that can be seen, touched, thought about, or uttered. Equally important, a link, like a subepisode, cannot be stand-alone. Links can only be created between two episodes. If either of these episodes is removed from the representation, then the links between them also must be destroyed.

The typical interactions which take place when a representation is created can be seen in **Figure 3-1.**

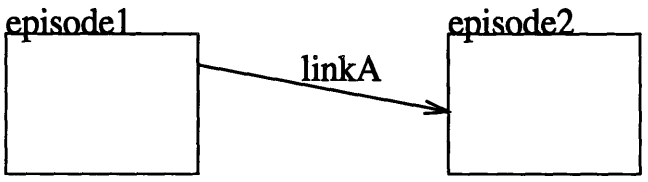

Figure 3-1: Sample scenario in which ***linkA*** describes the relationship between *episode1* and *episode2.*

In lexical notation, this can be thought of as

linkA *(episode1, episode2)*

This representation means that *episode1* has a relationship with *episode2* as specified by *linkA*. The different subepisodes are not shown but the reader should be realized that they are created at the terminations of the link and serve as the "anchors" for the link onto the episodes. This simple diagram when coupled with other scenarios can lead to a knowledge base from which other information can also be derived. In addition, the reader should also remember that this representation will reside in a networked environment and the episodes created can be shared with other users.

3.4 Participatory Semantics

Participatory semantics is the mathematics which will be used to derive information from the knowledge base, check for inconsistencies, and perform much of the negotiations that CATSI requires to

resolve the issues aforementioned. Symbols and operators such as '=' and '+' used in traditional mathematics will be replaced by the links that had been described above. In participatory semantics, "accountability is the norm" [Hewitt & Manning 94]. This means that representations of participation should be used to inform future participation.

Two tasks that these representations should perform are derivational calculus and inconsistency checking. Derivational calculus will use mathematical concepts and deductive inference to derive additional knowledge from the knowledge base. Consistency checking will be the means in which the representations are checked for correctness.

3.4.1 Derivational Calculus

Derivational calculus is the term which will be used to describe any deductive inference scheme used to create new relationships (links) with participants (episodes) based on previously defined relationships. Many of these schemes have parallels in mathematics and will thus be introduced by discussing the mathematical concepts from which it was conceived. One method which will be used in the derivation of further relationships is the idea of transitivity between links. In mathematics, transitivity can be viewed as in **Figure 3-2.**

Suppose:

$$a > b$$
$$b > c$$

Then it follows that:

$$a > c$$

Figure 3-2: Transitivity property in mathematics. Variables *a*, *b*, and *c* can be any variables that can be compared.

The transitivity here is illustrated by the greater-than symbol, but it also holds for the equality symbol and the less-than symbol. In participatory semantics, the following relation **(Figure 3-3)** also holds for certain links:

Suppose:

above *(a, b)*
above *(b, c)*

Then it follows that:

above *(a, c)*

Figure 3-3: Transitivity property in participatory semantics. Episodes *a*, *b*, and *c* can be any physical object in the real world.

This transitivity relation will be applicable only to relations such as ***contains***, ***before***, and others in which transitivity logically follows. Transitivity, like many other relations in derivational calculus, only involves links that are the same. Realizing this fact will allow for easier implementations of many algorithms which have been incorporated into VOPS.

From transitivity springs another similar rule which will be called negative transitivity. Slightly more complicated than the regular transitivity relation, negative transitivity involves a link and its negation link. In mathematics, this relationship can be seen as in **Figure 3-4**.

Suppose:

$a > b$
$b \not> c$

Then it follows that:

$a \not> c$

Figure 3-4: Negative transitivity in mathematics. Variables *a*, *b*, and *c* can be any values which can be compared.

Through a "negative" transitivity factor, as long as there is one negative relation, the remaining declarations will also be a negative relation. In participatory semantics, the following relation (**Figure 3-5**) also holds:

Suppose:

above *(a, b)*
~above *(b, c)*

Then it follows that:

~above *(a, c)*

Figure 3-5: Negative transitivity in participatory semantics. Episodes *a*, *b*, and *c* can be any physical object in the real world.

This negative transitivity will hold only for certain links and their negations. The ***contains*** relation will certainly be able to derive a negative transitivity relation; however, the ***before*** relation which although has a transitivity relation, will not necessarily have a negative transitivity relation. Working out a simple example should convince the reader why this is so. For example, if *3PM* is before *5PM* and *5PM* is not before *4PM*, then it is not true that *3PM* is not before *4PM*.

Another relationship involves the ***sameAs*** link and will be termed the reflexive rule. This is equivalent to the '=' operator in mathematics. Any two episodes which have a ***sameAs*** relation will be deemed the same. Thus, stems the following reflexive rule for the ***sameAs*** link (**Figure 3-6**):

Suppose:

sameAs *(a, b)*

Then:

sameAs *(b, a)*

Figure 3-6: The ***sameAs*** relation is reflexive. The lexical representation is shown above.

Hence, the following corollary to the reflexive rule, the equality rule, can also be derived:

Suppose:

sameAs *(a, b)*

Then:

For every relationship *i* and *j* in *a*, where
$link_i$ *(a, *)* and $link_j$ *(*, a)*
the same relationships will hold for *b* where
$link_i$ *(b, *)* and $link_j$ *(*, b).*

AND

For every relationship *i* and *j* in *b*, where
$link_i$ *(b, *)* and $link_j$ *(*, b)*
the same relationships will hold for *a* where
$link_i$ *(a, *)* and $link_j$ *(*, a).*

Figure 3-7: Corollary to reflexive rule from **Figure 3-6**. This corollary simply states that episodes *a* and *b* will look identical after using this rule.

This rule is termed the equality rule since, essentially, any two episodes with a ***sameAs*** connection will hold

the exact same connections to other episodes.

The two rules in **3-6** and **3-7** are only specific to the ***sameAs*** link. Other links, of course, will have their own rules which are independent of other links and rules which do depend on other links. These rules will also be able to derive their own relations from what is already given.

As the reader can tell, many rules stemming from mathematics can be derived along with these. Other logical rules that only make sense to specific links in the knowledge base can also be derived that are specific to one type of link or to multiple types of links. Since creating suitable algorithms for knowledge derivation using these rules take much time and effort, VOPS will only contain a small sample of derivational calculus rules.

3.4.2 Consistency Checking

In addition to derivation of other links from given links, participatory semantics may have some legality checks built-in as a feature. The checks can determine inconsistent or conflicting information in the system. In fact, the system that was built and will be described in the next chapter (VOPS) also contains rudimentary routines which will check for inconsistencies in representation. Consistency checks may or may not be feasible in a real-world scenario, but it is well worth taking a look.

One of the inconsistencies that may arise in participatory representation stems from the addition of negation links in the representation. For almost every relation, there can also be a negative relation. (The *lifeline* relation is an exception to the rule.) For example, just as

contains *(episodeA, episodeB)*

can exist, so can

~contains *(episodeA, episodeB)*

which means that *episodeA* does not contain *episodeB*. These two relations, if they were to reside in the same knowledge base, can create an inconsistency, detecting which will be the first step in resolving the issue. Negotiation will be introduced later to ask the participant that created the two scenarios to determine which one is correct.

Another common inconsistency can result if the user unwittingly creates a relation not knowing

about the previous links that had been created. Often times, the link and its negation are the culprits. For example, suppose the following relations had already been made:

before *(episode1, episode2)*
before *(episode2, episode3)*

.
.
.

before *(episodeN, episode(N+1))*

Then the userA decides to create this relation on his local system:

*~**before** (episode1, episode(N+1))*

The tilde (~) preceding ***before*** symbolizes a negation relation for the link. Graphically, this representation can be seen in **Figure 3-8.**

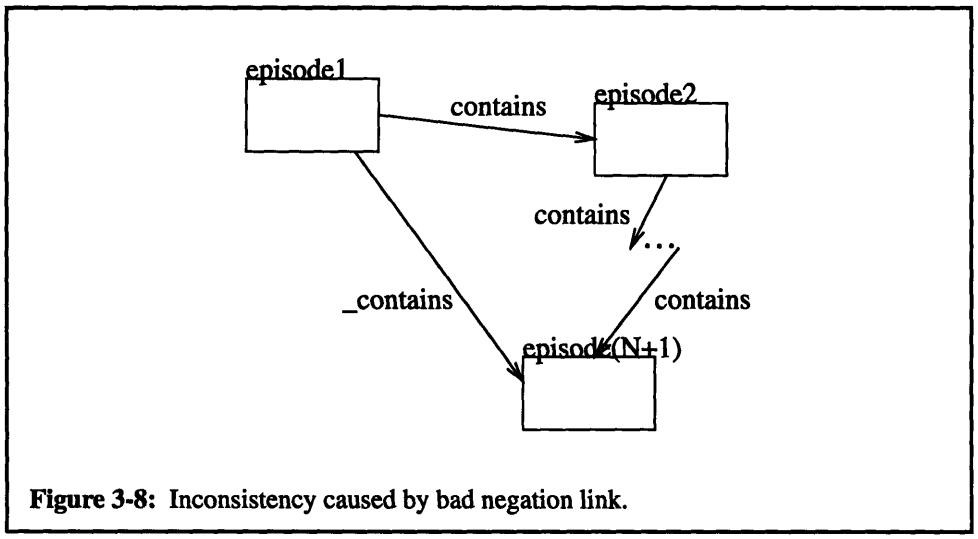

Figure 3-8: Inconsistency caused by bad negation link.

The user's representation of *episode1* and *episode(N+1)* has created an inconsistency. Again, the system has to signal that and inform the participants involved of the error.

Finally, another common inconsistency can result with different links that are not the negations of each other. For example, this is easily seen when one participant makes the representation:

before *(episodeA, episodeB)*

and another participant makes this representation:

after *(episodeA, episodeB)*

Intuitively, the existence of both links on both episodes will create an inconsistency in the way the system

interprets the relationship between *episodeA* and *episodeB*. In the English language, *before* and *after* have totally opposite meanings, and the system has to be aware of relations that are antonyms of each other to detect inconsistencies.

The three inconsistencies described above are but a fraction of the problems with which the system may have to cope. Any inconsistency will first require mediation and then negotiations, and it is beyond the scope of this thesis to explore all the possible ways of dealing with all of these situations. Instead, only a few of those checks and rules have been built into VOPS. The necessary question, however, is which, if any, inconsistency should be detected. Checking for inconsistencies may prove to be a huge burden and a waste of computing resources. Participatory semantics, like many knowledge representation systems, still does not have a sound solution to the inconsistency problem.

3.5 A Participatory Semantics Scenario Example

One way in which a reader may be able to understand how participatory semantics might work is to go through an example in which a detailed scenario is described, diagrammed, and resolved using participatory semantics. Suppose the following scenario of a child lost in the Museum of Science is given in **Figure 3-9**. This vignette demonstrates how users of TeleVisit will negotiate with the system to determine the location of the lost child. Users in this case will be the teachers who are carrying the portable computers.

Teacher1 and teacher2 are leading two groups around the Museum of Science, group1 and group2, respectively. John belongs to group1, and when the group goes to the computer exhibit, he is deemed lost by teacher1. After determining that John is missing, she creates a episodes and links which describe his activities, and asks teacher2 to do the same also. Now, it turns out that teacher2 saw John in the insects exhibit earlier. Having entered the entire scenario, the teachers hope to negotiate with the system to narrow down the search process. TeleVisit passes those two messages to the teachers, and the teachers use them to locate John. Teacher1 finally sees John where teacher sees2 saw John last, at the insects exhibit.

Figure 3-9: A lost child vignette. The situation is described, and the system follows a course of action to find the lost child.

There will be two users, two teachers (teacher1 and teacher2), that are leading groups of students

around the museum. The child in question is John, a mischievous kid, who enjoys science so much that he will go out of his way to participate in more exhibits and demonstrations. **Figure 3-10** shows the scenario that is currently residing in the system. The representations of both teachers have been combined to give an overall view of the scenario, and only the few events prior to John being lost have been included in the diagram. As can be seen from the diagram, John was last seen at the *Bird Exhibit* by teacher1, and he was last seen at the *Insects Exhibit* by teacher2. Also, John was lost by teacher1 when the group went to the *Computers Exhibit.*

Notice in **3-10** that the links between episodes are actually the relationships and connections between episodes that had been talked about earlier. ***Lifeline*** links are used to help trace the series of events that a participant has done.

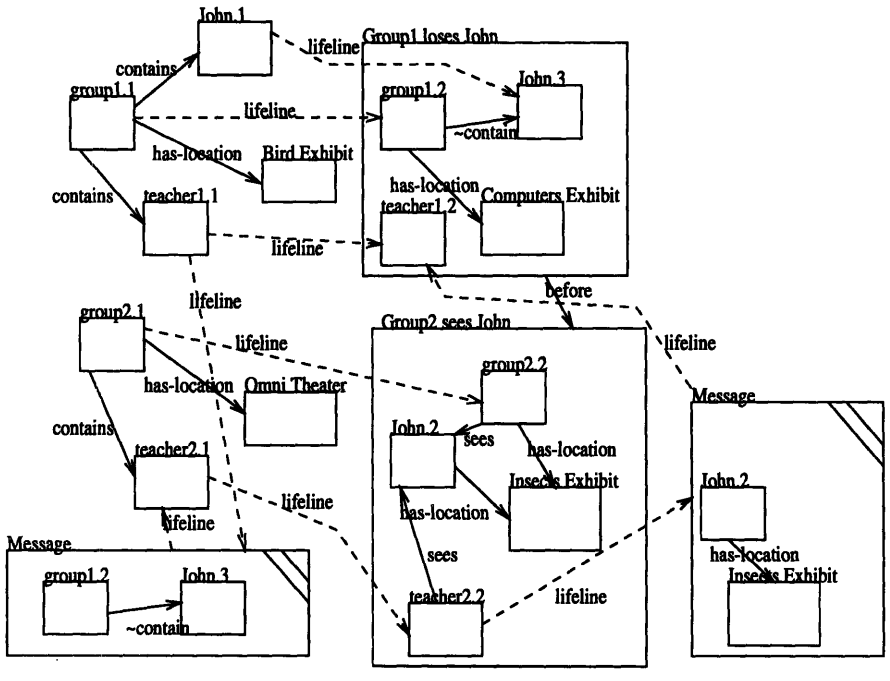

Figure 3-10: Scenario diagram taken from the lost child vignette.

Three other links are also present: ***contains***, ***~contains***, and ***hasLocation***. With this scenario and its associated episodes and links in memory, participatory representation will thus be used to create the resulting scenario in **Figure 3-11** which will narrow down John's location.

Figure 3-11 shows the original scenario diagram along with more episodes and links which the system has created with knowledge from its database. The episodes and links in dotted lines are the ones put in by the system. This is one way with which the system can take advantage of participatory semantics. Using semantics, teachers can pass messages to each other (such as the fact that "group1 does not contain John") and derive more relations (such as the fact that the *Group1 loses John* episode took place chronologically before the *Group2 sees John* episode). The system can then combine all the information and lead the teachers to finding the lost child who is sitting at the insects exhibit where he was seen last by teacher2.

Participatory semantics technology has thus been described in this chapter. The semantics and deductive inference techniques used have also been described. In addition, issues which participatory semantics hopes to resolve have been addressed. Finally, an example of how it can be used within the scope of TeleVisit has been demonstrated. Hopefully, these examples and concepts have created a clearer picture of participatory semantics. The next chapter will detail VOPS, which was built to help visualize participatory semantics.

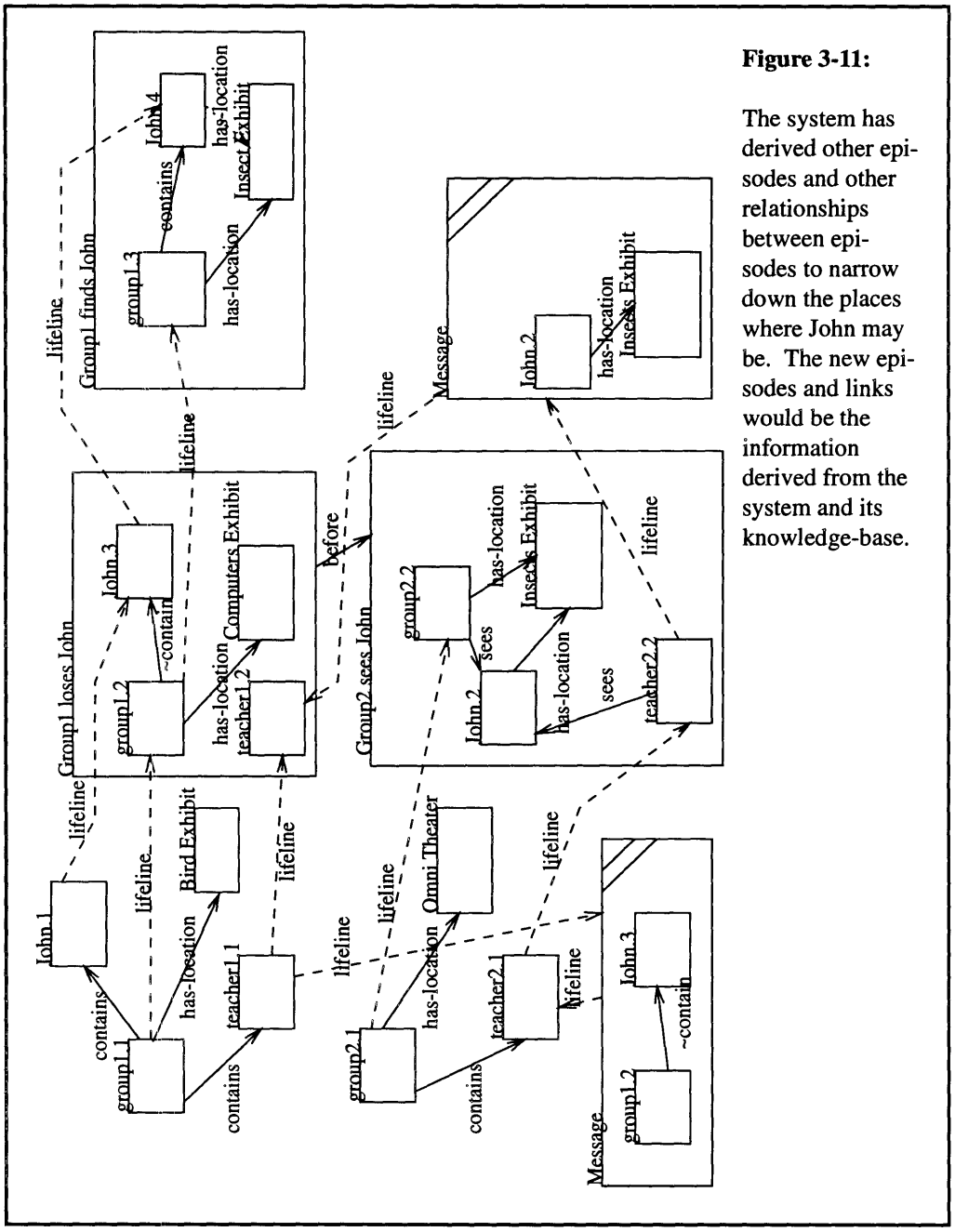

Figure 3-11:

The system has derived other episodes and other relationships between episodes to narrow down the places where John may be. The new episodes and links would be the information derived from the system and its knowledge-base.

Chapter 4 A Participatory Semantics Visualization System

A system has been designed and implemented which allows visual construction and analysis of the knowledge representation scheme that had been described in Chapter III. To recapitulate, this scheme is based on a set of objects called episodes that connect together with pre-defined links and subepisodes. Called VOPS, this design tool gives users a rudimentary set of tools and interfaces to create and modify various episodes that describe real-world scenarios. VOPS has the ability to derive information from what is already given and also check for simple inconsistencies that may come up in users' representations. A brief synopsis of this section follows. First, the kinds of technology that were used to build this system will be discussed. The reasoning behind each choice of technology will also be presented and clarified. A description of the class hierarchy will follow. All the major classes and subclasses will be discussed in detail with particular attention to the interfaces for each object. Some of the more interesting underlying implementations which support the interfaces will also be discussed. Finally, pseudo-code for the more interesting derivation and consistency checking routines will be given.

4.1 Technologies Employed

This entire system was designed and built around two leading edge systems that are available in industry. Microsoft's Visual C++ and Viewsoft's Utah are two Windows-based tools that enable quick designing, building, and debugging of programs. Visual C++ is the compiler front-end, and Utah is the graphical user interface builder.

As this software system expands, becomes more complicated, and requires coordination with the other components of the Museum of Science project, technologies that will enable easy porting of the system have to be used. Visual C++ presents a development environment that accomplishes all of the necessities enumerated above. It contains an integrated code-viewer, compiler, and debugger that provide rapid coding, building, and testing of the design tool. In addition, since Visual C++ is capable of running in the Windows NT environment, 16-bit Windows applications and 32-bit Windows NT applications can both be

built. This flexibility allows porting to many different types of systems. The mobile persistent object bases are envisioned to be running Windows 3.1 on mid-range machines such as Intel 486-class and low-end Pentium-class machines; the central object server will be running Windows NT as a server on high-end workstations that range from fast Pentium machines to RISC machines.

The other technology that has been extensively used is an object-oriented graphical user interface builder for Microsoft Windows called Utah. The reasoning for using Utah lies in the beauty with which it helps construct complicated user interfaces for application class objects. Using Utah, views are created for each object. Data members and member functions that require user interaction become Utah-specific and can easily be drawn using Utah allowing many choices for representing each member data or function. Utah will create all the necessary interactions (e.g., field updates, button pushes) for these members. Many views can also be created for each object allowing multiple ways to present an object to the developer and end-user. In addition, views can easily be embedded inside one another. A Utah-specific data member with its own particular view can be embedded within the view of the parent object, and any changes to the data member's view is automatically reflected in its parent. **Figure 4-1** shows the view for a dialog box which is used for editing an episode.

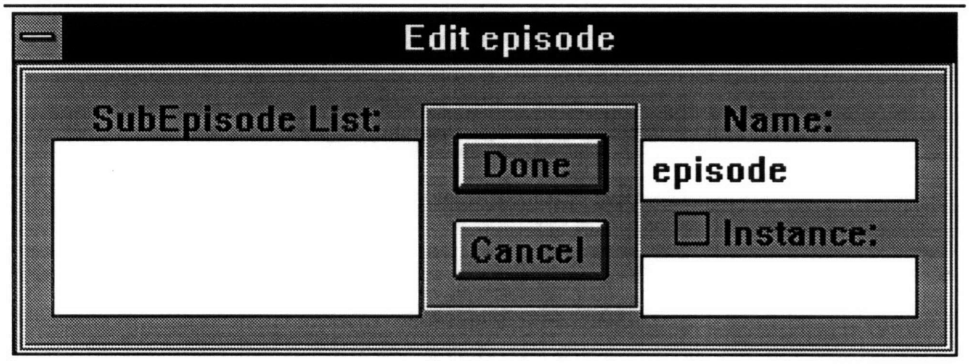

Figure 4-1: Dialog box created to edit data members of an episode.

After the views are created, Utah then generates the necessary Windows code, shells for each Utah-specific member function, and default constructors and destructors. Functional logic can then be inserted into each Utah-specific member function. Non Utah-specific member functions and member data can then

be added which perform more complex behind-the-scenes tasks. These members are termed non-interactive in that they do not require user interaction. Other application objects can also be created and hooked into the generated code to help implement other functionalities. For example, a new class of command objects has been created and derived from Utah command objects to assist in the drawing of shapes on the screen. These last steps all happen independently of the first steps. Thus, changes to one aspect of the code will not affect others. More views can be created and added without affecting the implementation. And different implementations of each class can be created without affecting the graphical user interface.

These two technologies were thus chosen for their modularity in designing class objects, longevity for use in future operating systems, portability to different architectures, and orthogonality in interface and implementation code. The modularity and orthogonality provided by Utah allow subtle and complicated changes to the graphical user interface without affecting the underlying implementation of the objects. The Windows environment promises to be a platform which will always be well-supported. Visual C++ is capable of running on Windows 3.1 and Windows NT on Intel 80x86 platforms. And Windows NT is able to run on non-Intel architectures such the DEC Alpha and Power PC platforms. So with Visual C++, development can be done with minimal concern about portability and longevity issues.

Thus, building of VOPS using the representation scheme described previously will not involve heavy use and knowledge of the Windows API and learning of Windows constructs. More attention can be paid to the design details of the classes which implement the participatory semantics technology.

4.2 Class Hierarchy

For this knowledge representation scheme, **Figure 4-2** shows the class hierarchy that is to be used. The primary base class will be called an episode. Derived from an episode is a link which serves as a connection between various episodes. Other links are derived from the link class. A special type of link is the ***lifeline***. A ***lifeline*** serves to connect episodes that actually describe the same object as it makes transitions through different stages of its life. The ***lifeline*** link will explained in detail in section **4.3.2** when the link subclass is discussed. Other links connect different episodes together to form different relationships. For example, there is a ***contains*** link that describes which objects are contained within which others and there is

a ***before*** link which describes temporal relationships between episodes.

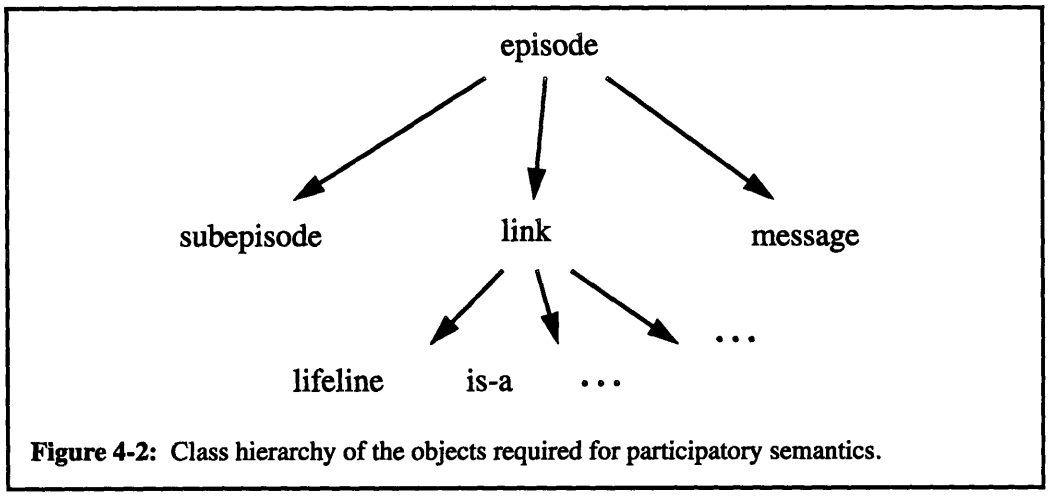

Figure 4-2: Class hierarchy of the objects required for participatory semantics.

Another class derived from the episode is a subepisode. Subepisodes serve as the anchors upon which links attach to in a given episode. The next figure (**Figure 4-3**) gives an example of a typical relationship between episodes, links, and subepisodes. From this figure, it can be seen that links originate and terminate on subepisodes. Subepisodes reside in episodes. Thus episodes will just be containers for slews of subepisodes.

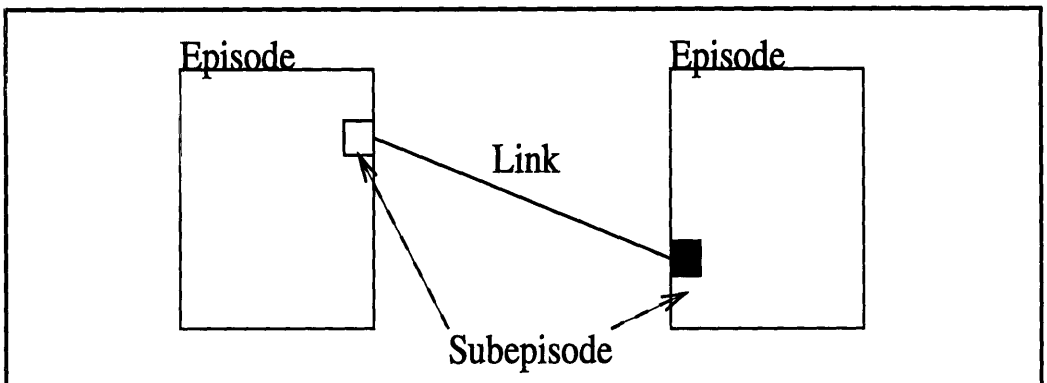

Figure 4-3: Subclass relationships in participatory semantics.

The final class derived from an episode is the message class. A message is a special type of episode that travels from one episode to another. More discussion of what a message is will follow as concrete examples of how a scenario can be described using episodes are shown. **Figure 2-1** from **Chapter 2** contains a good diagram of the uses for messages in TeleVisit. Participants can send and receive messages from

the system which in turn helps to interpret those messages.

4.3 Class Descriptions

4.3.1 Episode Base Class

The primary base class used in this system is an *episode*. The code in **Appendix B** refers to an *episode* as a *kRepEpisode*. For various reasons, all the class names that are mentioned in this paper will be prefixed by *kRep* in the C++ code. *kRep* stands for *knowledge representation.* Furthermore, since this system was developed with all the graphical user interface objects provided by Utah, all application objects will inherit from *EosObject*. But, for the sake of this discussion, an *episode* will be spoken of as ***the*** base class and is not derived from anything.

Each episode has a *name* associated with it and an *instance* identifier to distinguish episodes of the same *name* but with different information about the outside world. For example, two episode of the type "dog" can have instances "hungry" and "fed", which can further differentiate the "dog" episode. Use of the *instance* identifier is completely optional but is recommended when creating ***lifeline*** links between episodes. In addition, each episode has a *shape* to allow itself to be viewed graphically. The default shape of an episode is a rectangle. Each episode also contains an *episodeList*, which is a list of episodes. This list can contain other episodes and subepisodes. Subepisodes will be explained later in this chapter. And, finally, each episode will also contain a reference back to another episode which will be deemed the *creatorEpisode*. Every episode that has been derived from another episode with the deductive inference logic built into VOPS will have a non-null *creatorEpisode*. When the *creatorEpisode* is deleted, all the episodes that were created by that one will also be destroyed since they could not have existed without the existence of the *creatorEpisode*.

The simplified header file for the episode base class is shown here. Member data, interfaces, and important implementation functions are presented here also. (The code listings in **Appendix B** all use Utah types in class objects to specify data members, return values, and function arguments.)

```
class kRepEpisode
{
        private:
```

```
        EosString name;
        EosString instance;
        kRepShape *shape;
        EosBoolean selected;
        EosBoolean showInstance;
        kRepEpisodeList *episodeList;
        kRepEpisode *creatorEpisode;
protected:
        EosBoolean selfControl;
        EosBoolean parentDeletable;
public:
        //Member access functions
        EosString getName() { return name; }
        kRepShape *getShape() { return shape; }
        kRepEpisode *getCreator() { return creatorEpisode; }
        EosBoolean isSelected() { return selected; }
        void parentIsDeletable(EosBoolean);
        void selfControllable(EosBoolean);

        // Utah-specific member access function
        virtual void edit(kRepView &);

        // Graphic routines for drawing
        virtual void draw(kRepView &);
        virtual void drawFeedback(kRepView &);

        // Bounds checking interfaces
        virtual EosBoolean containsPoint(EosPoint &);
        virtual EosBoolean containsEpisode(kRepEpisode *);
        virtual void addContainedEpisodes(kRepEpisodeList *);
        kRepEpisode *findContainerEpisode(kRepEpisodeList *);

        // Interfaces for other episode objects and classes derived from episode.
        virtual void removeAssociations(kRepEpisodeList *, boolean);
        virtual EosBoolean isParentOf(kRepEpisode *);
        kRepEpisode *findParent(kRepEpisodeList *);
        virtual void addEpisode(kRepEpisode *);
        virtual void removeEpisode(kRepEpisode *);
        virtual void removeAssociations(kRepEpisodeList *, EosBoolean);
        virtual void findID(tools, direction, kRepEpisodeList *);
        virtual boolean allowSubEpisode(kRepSubEpisode *, kRepEpisodeList *);
}
```

All the interfaces and implementations for the episode class are listed above. Everything underneath the public header is an interface to the episode class whereas all the others (protected and private) can be considered implementation details. The interface functions allow interaction between an episode and other objects. General descriptions of all the interfaces follow. More detailed descriptions can be found in **Appendix A**.

Other member data shown in the header are just booleans to be used as flags. The two important,

non Utah-specific, ones are *selfControl* and *parentDeletable*. For most episodes, the former will be set to true and the latter set to false. Episodes generally have control over their fates. Thus if an episode's *self-Control* flag is set to true, his parent will not delete him when the parent itself is being deleted. Subepisodes, on the other hand, will have their *selfControl* flag set to false to ensure destruction once their parent has been destroyed. The other flag, *parentDeletable*, ensures the destruction of a parent when its child is being destroyed. One example of the need for this is in the destruction of subepisodes that lie in episodes. They should not be able to destroy their parent. However, subepisodes whose parents are links should inform parents to destroy themselves also. As seen in **Figure 4-3** from above, links cannot be left dangling in space when their subepisodes are deleted.

Four interfaces allow access to member data

```
EosString getName() { return name; }
kRepShape *getShape() { return shape; }
kRepEpisode *getCreator() { return creatorEpisode; }
virtual void edit(kRepView &);
```

The **getShape**, **getName**, and **getCreator** functions do exactly what their names imply. The **edit** function also accomplishes the same task by popping up a dialog box which can be edited by the user.

Other major interface functions can be broken down to two sets. One set of functions deals primarily with screen manipulation of episodes. The other set involves the deeper relationships between episodes that transcend any screen boundaries. These interfaces define what goes on within and between episodes.

First, routines are needed to place the episodes on the screen.

```
virtual void draw(kRepView &);
virtual void drawFeedback(kRepView &);
```

The **draw** routine draws the shape of the episode in the view specified in the argument. The **drawFeedback** function is similar to draw but is used to display the shape as it is being drawn.

Still other interfaces are used to check for constraints of episodes on the screen.

```
virtual EosBoolean containsPoint(EosPoint &);
virtual EosBoolean containsEpisode(kRepEpisode *);
```

The **containsPoint** interface checks to see if a point is within the bounds of the shape defined by the episode. The **containsEpisode** interface accomplishes the same task except it takes an episode as an argument and

checks that the shape for the argument being called lies within the bounds of the shape of the object. Both functions return a true value if the episode contains the specified argument. Note that both routines use purely geometrical concepts in order to check for containment. **Figure 4-4** shows an example of a scenario diagram that can cause conflicts when determining containment. Point **E** will be true when being checked for containment inside episodes **A, B,** and **C**. Similarly, episode **C** will also return true when being checked for containment inside episodes **A** and **B**. Episode **D** and point **F**, however will have none of those problems with multiple episodes containing the same episode or point. These containment problems will just have to be resolved with specialized functions which will be discussed next.

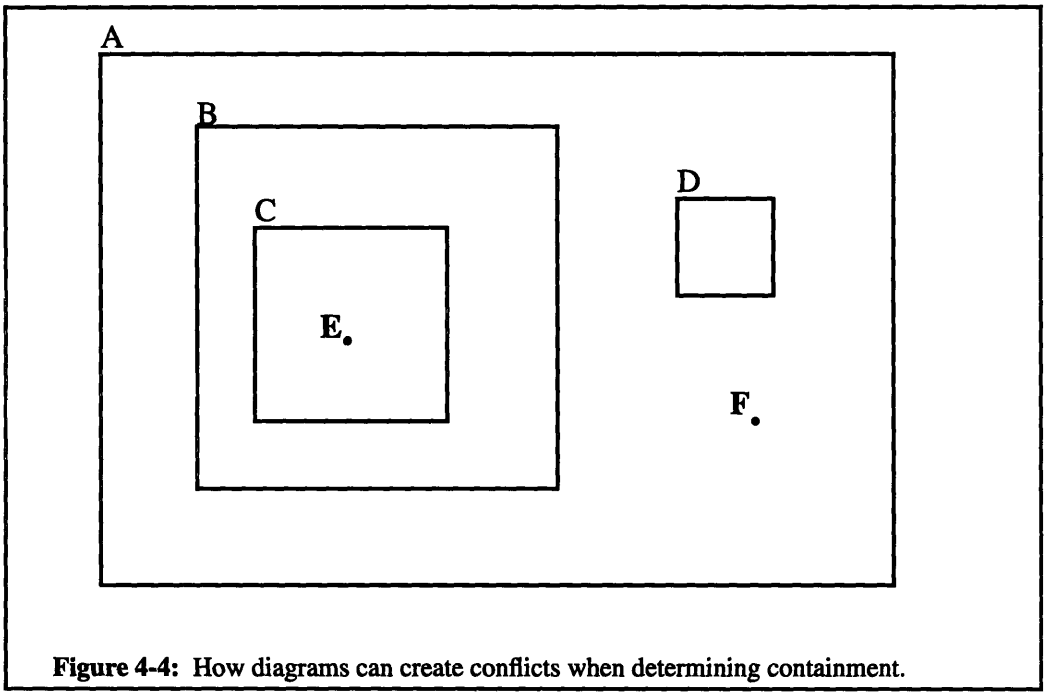

Figure 4-4: How diagrams can create conflicts when determining containment.

The next two functions use the **containsEpisode** function to perform specific tasks.

virtual void ***addContainedEpisodes***(kRepEpisodeList *);
kRepEpisode ****findContainerEpisode***(kRepEpisodeList *);

These interfaces that also deal with screen manipulation and boundary checking are **addContainedEpisodes** and **findContainerEpisode.** The former finds all the children of this episode. The term children implies all the immediate episodes that are physically included within the boundaries of an episode (that is called the parent). This means that any episodes included within the children will not be added to the episode list of the parent episode. For example a call to **A.addContainedEpisodes** will result in episode **A**

adding episodes **B** and **D** to its subepisode list. A call to **B.addContainedEpisodes** will result in **B** adding **C** to its subepisode list. The other interface, **findContainerEpisode**, determines the episode that immediately contains this episode. This is the geometrical way of finding the immediate parent to an episode. A call to **C.findContainerEpisode**, for instance, will return episode **B** which is the immediate geometrical parent of episode **C**.

Direct manipulation of episodes that does not involve screen geometries utilizes functions that are simpler to maintain.

```
virtual EosBoolean isParentOf(kRepEpisode *);
kRepEpisode *findParent(kRepEpisodeList *);
```

The **isParentOf** function is similar to the **containsEpisode** function. It simply takes an episode and checks to see if the episode is in the sub episode list that resides in the parent. The other interface, **findParent**, is analogous to **findContainerEpisode.** It uses **isParentOf**, traverses down the list of episodes supplied as the argument, and attempts to find the parent of the episode

Two more interfaces, **removeEpisode** and **addEpisode**, also allow direct manipulation of the subepisode list contained within an episode.

```
virtual void addEpisode(kRepEpisode *);
virtual void removeEpisode(kRepEpisode *);
```

The former directly removes a given subepisode from an episode's subepisode list if the episode exists, and the latter adds an episode to the end of the episode's subepisode list.

A **removeAssociations** function has also been included to be used as a supplement to the regular delete operator supplied by C++.

```
virtual void removeAssociations(kRepEpisodeList *, EosBoolean);
```

Use of this function is required because each object created in VOPS will be referred to from many other objects. Each object needs to remove all references to itself when it needs to be deleted. So the **removeAssociations** interface removes all references of this object from a list of episodes presented in the argument. If the second argument is true, then **removeAssociations** will also destroy the object in memory. In the base implementation, **removeAssociations** iterates down the episode list of itself, calling **removeAssociations** on every subepisode. This will destroy any links that this episode may also have.

Another interface to the episode class is **findID**.

virtual void ***findID***(tools, direction, kRepEpisodeList *);

This function takes three arguments, the kind of link that the subepisode refers to (tools), the direction that a subepisode is pointing (direction), and a list of episodes to store the results of the search. When finished, **findID** will find all the subepisodes in its episode list that match the ID and direction specified, and it will store the results in the episode list presented in the third argument.

Finally, there is the **allowSubEpisode** interface.

virtual boolean ***allowSubEpisode***(kRepSubEpisode *, kRepEpisodeList *);

It takes a subepisode as argument and a episode list (usually the set of all the episodes in a scenario) and returns a boolean specifying whether a given subepisode is legal to be created within itself. This interface will be discussed in more detail in the subepisode subclass.

An episode is thus an object that contains other episodes, which will be referred to as subepisodes. The interface described above for an episode contains all the functions needed to manipulate itself and all its subepisodes. Since this is a base class, many functions are virtual to allow subsequent subclasses to reinstantiate them.

4.3.2 Link Subclass

A link is derived from an episode. It is represented graphically by a line (**Figure 4.5a**) but when expanded looks just like an episode as in **Figure 4-5b**. The squares inside the expanded link episode signify subepisodes. The filled square denotes the termination of the line, while the unfilled square denotes the origination of the line just as in **Figure 4-1** above. Links are used to connect different episodes together.

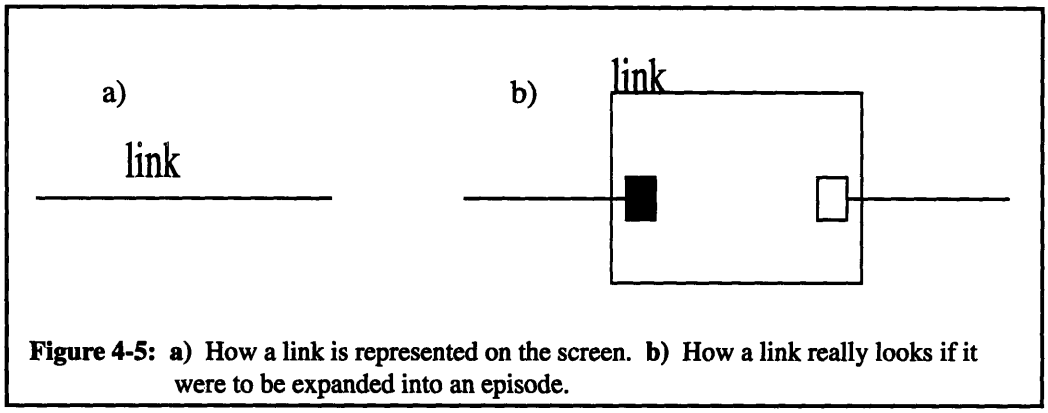

Figure 4-5: a) How a link is represented on the screen. **b)** How a link really looks if it were to be expanded into an episode.

The class name for a link is *kRepLink*, and the simplified header file is listed below.

```
class kRepLink : public kRepEpisode
{
        public:
                // Graphic routines for drawing
                virtual void draw(kRepView &);
                virtual void drawFeedback(kRepView &);

                // Mutator functions
                virtual void setFromEpisode(kRepEpisode*);
                virtual void setToEpisode(kRepEpisode*);

                // Member access functions
                virtual kRepEpisode *getFromEpisode();
                virtual kRepEpisode *getToEpisode();

                // Bounds checking interfaces
                virtual EosBoolean containsPoint(const EosPoint &);
                virtual EosBoolean containsEpisode(kRepEpisode*);

                // Interfaces for other episode objects and classes derived from episode.
                virtual boolean isParentOf(kRepEpisode*);
                virtual boolean removeAssociations(kRepEpisodeList*, EosBoolean);
                virtual void setupLink(kRepSubEpisode*, kRepSubEpisode*, tools,
                                        kRepEpisodeList*);
                virtual void derive(kRepEpisode*, kRepEpisode*, tools,
                                        kRepEpisodeList*);
                virtual EosBoolean allowSubEpisode(kRepSubEpisode*, kRepEpisodeList*);
                virtual void makePretty(EosPoint&, EosPoint&, kRepEpisode*,
                                        kRepEpisode*);
}
```

Most of the interfaces that are specified for episodes have to be re-implemented for the link sub-class. A link is a special type of episode that is limited to having two subepisodes (As shown in the above figure). One subepisode contains a reference to the origination of the link while the other references the des-

tination of the link. These two subepisodes can be accessed with **getFromEpisode**, **getToEpisode**, **setFromEpisode**, and **setToEpisode**.

virtual void ***setFromEpisode***(kRepEpisode*);
virtual void ***setToEpisode***(kRepEpisode*);
virtual kRepEpisode ****getFromEpisode***();
virtual kRepEpisode ****getToEpisode***();

Since the symbol for denoting a link in this system is a line, the functions for **draw** and **drawFeedback** also require re-implementation for a link. In addition, the **containsPoint** and **containsEpisode** functions for direct manipulation of episodes on the screen have to be changed. The **containsPoint** function has been implemented to check if a point is within a certain number of pixels of the line. The interface, **containsEpisode**, will always returns false since, as shown on the screen, lines cannot physically include any other object.

The **isParentOf** and **removeAssociations** functions for a link are simpler than the ones for episodes since each function only has to check for two items in its subepisode list.
They have thus been reimplemented for speedier computation.

New interfaces that are specific to links are **setupLink**, **derive**, and **makePretty**.
The **setupLink** interface takes four arguments: two subepisodes, a "from" subepisode and a "to" subepisode; a linkID (tools), which specifies the type of link being created; an episode list, which is usually the set of all episodes in the scenario.

virtual void ***setupLink***(kRepSubEpisode*, kRepSubEpisode*, tools,
kRepEpisodeList*);

This interface attempts to setup a link between the two subepisodes, referencing all subepisodes correctly to one another.

Another interface, **derive**, also takes four arguments: two episodes, a *from* episode and a *to* episode; a linkID (*tools*), which specifies the type of link being created; an episode list, which is usually the set of all episodes in the scenariod.

virtual void ***derive***(kRepEpisode*, kRepEpisode*, tools,
kRepEpisodeList*);

Derive attempts to derive a link of the type specified by *tool* between the two episodes passed in as arguments. It also checks to see if a link of exactly that type already exists between the two episodes. If so, then

it will not create the link.

The **makePretty** interface is a screen manipulation routine that is called to place subEpisodes on the borders of their parent episodes.

virtual void ***makePretty***(EosPoint&, EosPoint&, kRepEpisode*, kRepEpisode*);

Finally, the **allowSubEpisode** interface for a link has to be redefined to always return a true value. Links, in the most basic implementation do not contain any subepisodes in itself that can determine the legality of other subepisodes. When a subepisode is created within link using **setupLink** or **derive**, legality is always insured. And since no other function will create any subepisodes within a link, the **allowSubEpisode** interface should always return a true value.

Specific links with special functionality have been created within the system. These links all derive from the kRepLink class and are enumerated below.

kRepLifeline
sameAsLink
notSameAsLink
containsLink
notContainsLink
beforeLink
notBeforeLink
afterLink
notAfterLink
isALink
notIsALink

All the links that are available within the system provide for temporal or spatial relationships among episodes. Using these links, one can describe many real-world scenarios. More links can of course be added to the system to provide other descriptions. These will just have to be created within Utah with its functionality added in and compiled using Visual C++. For convenience, a ***null*** link has also been provided that requires no legality checking. In addition, each type of link (except for a ***lifeline***) has a negative complement so that a user can explicitly specify a "not" relation. For example, a ***notIsA*** link between an episode prevents any further derivation of an ***isA*** link that might be created later. Examples will be given which will discuss the necessity of converse relations.

The one link that warrants special attention is the ***lifeline*** link. This link describes an object that is going through different phases of its "life" with different properties about itself. A simple example is shown

in the next figure (**4-6**) of an episode with a ***lifeline***.

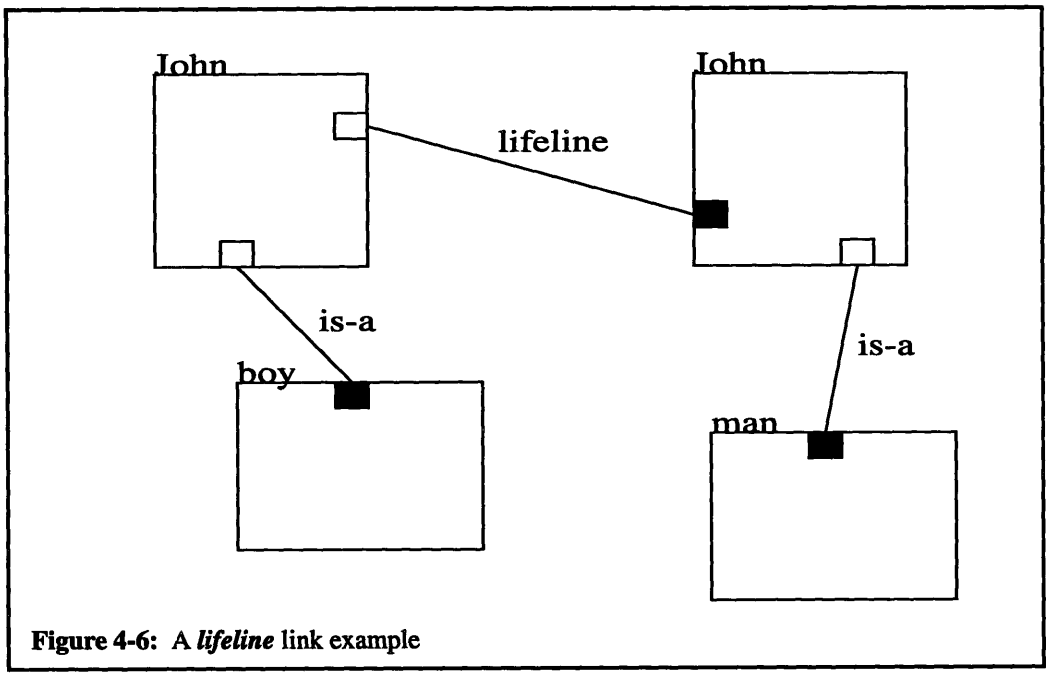

Figure 4-6: A ***lifeline*** link example

In the figure John is initially a boy, but he becomes a man when he traverses the lifeline. This type of link will be very useful, for example, when a person's movement is tracked within the Museum of Science.

Like that of episodes, the data structure for links appears to be merely a container for other episodes, except that the types of links that exist in this system will always contain a maximum of two subepisodes. Links contain no logic to determine legality nor any to derive further knowledge about scenarios residing within VOPS.

4.3.3 Subepisode Subclass

The next class derived from episode, a subepisode, will incorporate all the features of consistency checking and knowledge derivation. Subepisodes can be thought of as anchors within episodes that hold the endpoints of links. Subepisodes contain knowledge concerning the entire system and can query other subepisodes for information.

Subepisodes are very complex episodes. They contain many other interfaces in addition to the interfaces that are reimplemented from the episode base class. The simplified header file is shown below.

```
class kRepSubEpisode : public kRepEpisode
{
        private:
                tools linkID;
                direction dir;
        protected:
                // Consistency checking routines
                EosBoolean lifelineAllowable(kRepSubEpisode*, kRepEpisodeList*);
                EosBoolean containsAllowable(kRepSubEpisode*, kRepEpisodeList*);
                EosBoolean isAAllowable(kRepSubEpisode*, kRepEpisodeList*);
                EosBoolean sameAsAllowable(kRepSubEpisode*, kRepEpisodeList*);
                EosBoolean beforeAllowable(kRepSubEpisode*, kRepEpisodeList*);
                EosBoolean afterAllowable(kRepSubEpisode*, kRepEpisodeList*);
                EosBoolean notContainsAllowable(kRepSubEpisode*,
                                        kRepEpisodeList*);
                EosBoolean notIsAAllowable(kRepSubEpisode*, kRepEpisodeList*);
                EosBoolean notSameAsAllowable(kRepSubEpisode*,
                                        kRepEpisodeList*);
                EosBoolean checkTransitive(kRepSubEpisode*, kRepEpisodeList*);
                EosBoolean checkReflection(kRepSubEpisode*, kRepEpisodeList*);
                EosBoolean checkDependence(kRepEpisodeList *, kRepSubEpisode*,
                                        tools);

                // Derivational routines
                virtual void deriveTransitive(kRepEpisodeList *);
                virtual void deriveNegativeTransitive(kRepEpisodeList *);
                virtual void deriveSameAs(kRepEpisodeList *);
        public:
                // Member access functions
                direction getDirection();
                direction getOppositeDirection();
                tools getLinkID();
                tools getConverseLink();
                kRepSubEpisode *getReferenced();

                // Interfaces for other episode objects and classes derived from episode.
                virtual boolean findDuplicate(kRepEpisode *, kRepEpisode *,
                                        kRepEpisodeList *);
                virtual void removeAssociations(kRepEpisodeList*, boolean);
                virtual boolean isParentOf(kRepEpisode*);
                virtual boolean allowSubEpisode(kRepSubEpisode*,
                                        kRepEpisodeList*);
                kRepSubEpisode *getTermination(kRepEpisodeList *);
                virtual void deriveNew(kRepEpisodeList *);
                virtual void initLink(kRepLink **, tools);
}
```

The interface for the subepisode class is noticeably larger, and it contains many more implementation functions. One interface which is redefined in the subepisode class is **removeAssociations**. This implementation of **removeAssociations** needs to be different. As seen in **Figure 4-3**, once a subepisode is

deleted, the corresponding link which references the subepisode also has to be deleted. In addition, the terminating episode at the other end of the link has to be removed from its parent also. All of these steps require a more complicated implementation of a subepisode. The code for this can be seen in **Appendix B**.

Another interface that has been redefined is the **isParentOf** function. A subepisode will not contain any children episodes in this implementation. In its episode list, the subepisode object only contains one element which references to another subepisode which has its own parent. So the **isParentOf** interface for a subepisode will just return a false value.

The rest of the interface to a subepisode is specific to the subepisode subclass. The **getReference** function returns just the episode that it references.

kRepSubEpisode ***getReferenced***();

The other function, **getTermination**, finds the subespisode that is being referenced at the other end of a link.

kRepSubEpisode ***getTermination***(kRepEpisodeList *);

This key function will allow traversal up and down chains of episodes that are connected by links.

The implementation of a subepisode also requires the addition of two variables. One variable, *linkID*, is an enumeration that stores the type of link that the subepisode is anchoring. The other, *dir*, is also an enumeration which stores the direction that the subepisode is pointing. Three directions are possible: *origination*, *destination*, and *none*. Non-directional links, such as ***sameAs***, will have subepisodes with their *dir* variable set to *none*. Other subepisodes that are referred to by uni-directional links will have their *dir* variable set to origination or destination. The choice of direction will be determined by whether the subepisode is at the head or tail end of the link. Interface functions which allow access to the new variables are **getLinkID**, **getDirection**, **getOppositeDirection**, and **getConverseLink**.

direction ***getDirection***();
direction ***getOppositeDirection***();
tools ***getLinkID***();
tools ***getConverseLink***();

These functions should be self-explanatory, but detailed descriptions can also be found in **Appendix A**.

The rest of the interface and implementation functions contain all of the logic in knowledge derivation and knowledge checking. The function **allowSubEpisode** takes a subepisode as its first argument and

attempts to call one of the legality checking functions depending on the type of link that the subepisode was created using.

```
virtual boolean allowSubEpisode(kRepSubEpisode*,
                                kRepEpisodeList*);
```

These functions are named by taking a link type and suffixing it by the word "Allowable".

```
EosBoolean lifelineAllowable(kRepSubEpisode*, kRepEpisodeList*);
EosBoolean containsAllowable(kRepSubEpisode*, kRepEpisodeList*);
EosBoolean isAAllowable(kRepSubEpisode*, kRepEpisodeList*);
EosBoolean sameAsAllowable(kRepSubEpisode*, kRepEpisodeList*);
EosBoolean beforeAllowable(kRepSubEpisode*, kRepEpisodeList*);
EosBoolean afterAllowable(kRepSubEpisode*, kRepEpisodeList*);
EosBoolean notContainsAllowable(kRepSubEpisode*,
                                kRepEpisodeList*);
EosBoolean notIsAAllowable(kRepSubEpisode*, kRepEpisodeList*);
EosBoolean notSameAsAllowable(kRepSubEpisode*,
                                kRepEpisodeList*);
```

Hence, if a subepisode, *this*, is checking if another subepisode, *test*, that refers to a contains link is legal, the interface, **allowSubEpisode**, of the subepisode *this* will call the function **containsAllowable** with *test* as its first argument.

The **allowSubEpisode** interface is only called when the user is modifying scenarios. It will send a message (the *kRepMessage* subclass will be described later) to the creator of the errant link (in this case, messages will always be sent directly to the screen since VOPS is a single user system.) informing him/her of an inconsistency.

The other interface which allows the subepisode to directly modify a scenario is the **deriveNew** function.

```
virtual void deriveNew(kRepEpisodeList *);
```

This interface is also called after the user modifies the system by adding a link. **DeriveNew** will call as many of the implementation functions, **deriveTransitive, deriveNegativeTransitive**, and **deriveSameAs** as it can based on the type of link created.

```
virtual void deriveTransitive(kRepEpisodeList *);
virtual void deriveNegativeTransitive(kRepEpisodeList *);
virtual void deriveSameAs(kRepEpisodeList *);
```

Those three implementation functions for information derivation were described in detail in **Chapter 3**. The source code for their implementation can be found in **Appendix B**. The functionalities in those three imple-

mentation include the ability to derive transitive links and negative transitive links and derive duplicate links for an episode once a ***sameAs*** relation is established between episodes.

4.3.4 Message Subclass

The final class that is derived from the kRepEpisode base class is the kRepMessage class. A message is defined as anything information that is sent from one episode to another. The simplified header file is shown below.

```
class kRepMessage : public kRepEpisode
{
        private:
                EosString message;
        public:
                virtual void send();
                virtual void send(kRepDocument*);
}
```

A message works similar to a link. A message's episode list contains a list of episodes which becomes a list of the sender and receivers of the message. The first element of the episode list of a message contains the address of the originator, and all subsequent elements are the episodes that are receiving the message. A message when told to send itself will send itself to all the destinations listed in the list. Because the system being created right now is a single user system, all messages will be relayed back to the user.

For VOPS, messages are only used as warnings. When the user creates a seemingly illegal link, the subepisode which catches the inconsistency will send a message to the user stating the problem. The user is responsible for correcting the inconsistency if he/she wishes. The system will then assume that every representation that was made is consistent.

4.4 Algorithms for Consistency Checking and Deductive Inference

This section will outline some of the algorithms that were used to build the "smarts" into the system. Note that all of these algorithms have been incorporated into the respective implementation functions of the kRepSubEpisode subclass. A brief description and diagram of the problem that the function is trying to resolve will be given for each algorithm, and then pseudo-code will be presented. The actual C++ code

can again be seen in **Appendix B.**

4.4.1 CheckDependence Routine

In many cases, one type of link is dependent on another type for its legality. This means that this link will cause an inconsistency when it is created with the presence of the link with which it is dependent. A simple example that was given in **Chapter 3** will be redisplayed here in **Figure 4-7.** A common check for inconsistency with the **checkDependence** routine is among links and their negative couterparts. In this scenario, when the *~contains* link is created between *episode1* and *episode(N+1)*, checkDependence will need to catch the dependence between *contains* and *~contains.*

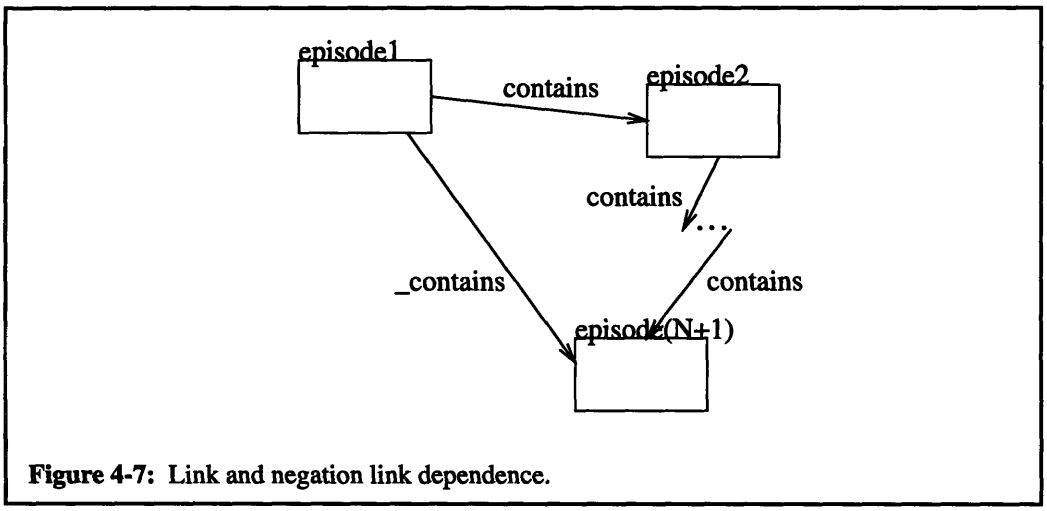

Figure 4-7: Link and negation link dependence.

The pseudo-code for **checkDependence** is given in **Figure4-8** below. This routine is called on the subepisodes which were created from adding a new link (in this case, ***~contains***), and it takes in four variables, the *list* of all the episodes in the scenario; an *exclude* episode which will always be skipped in the search; the link that the newly-created link is *dependent* upon, and finally the *direction* of the dependence. Using **Figure 4-7**, it can be seen that the idea is that **checkDependence** is checking for dependencies between the newly-formed ***~contains*** link (whose subepisodes call this routine) and any related ***contains*** links. It will thus iterate up all the episodes which has a ***contains*** relation with *episode(N+1)* until it finds an episode that matches the terminating end of the

```
kRepSubEpisode::checkDependence(list, exclude, dependent, dir)
    parent = findParent(list)
    tmpList->addObject(this)
    while(true)
        while(tmpList != NULL)
            get current subepisode from tmpList
            if(current != exclude)
                find termination of current
                tmpParent = termination->findParent(list);
                if (parent == tmpParent)
                    return true
                else
                    tmpParent->findID(exclude->getLinkID(),
                                      cur->getOppositeDirection(),
                                      workingList)
                    tmpParent->findID(dependent, dir, workingList)
            tmpList = tmpList->next()
        if (workingList==NULL)
            return false;
        else
            tmpList = workingList
            workingList = NULL
```

Figure 4-8: Pseudo-code for the **checkDependence** routine. The C++ implementation of this code can be seen in **Appendix B.**

~contains link, which in this case will be *episode1*. The pseudo-code given above, and all subsequent pseudo-codes, will use the interfaces for the different classes specified previously in this chapter. In addition, C++ types and calls will also be used.

4.4.2 DeriveTransitive Routine

The **deriveTransitive** routine attempts to derive any transitive links that may result after the addition of a link that supports transitivity. A diagram of this situation can be seen in **Figure 4-9** below. The solid lines denote the links that were given to the system. Dotted lines denote links that have been derived by the system.

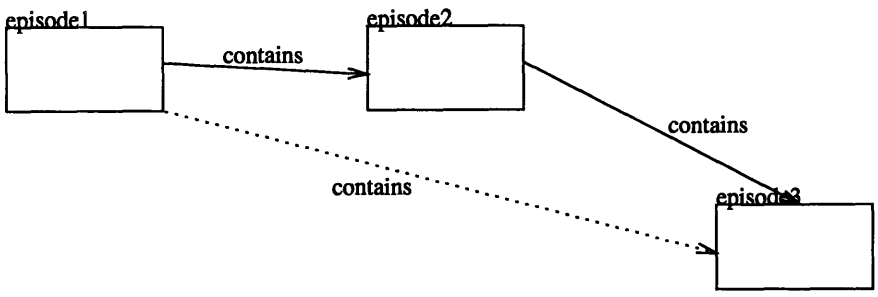

Figure 4-9: Diagram of situation in which transitivity is derived.

The pseudo-code for **deriveTransitive is shown** in **Figure 4-10** below.

```
kRepSubEpisode::deriveTransitive(list)
    parent = findParent(list)
    refParent = getTermination(list)->findParent(list)
    parent->findID(linkID, getOppositeDirection(), tmpList)
    while(true)
        while(tmpList != NULL)
            get current subepisode from tmpList
            find termination of current
            tmpParent = termination->findParent(list)
            if (tmpParent == refParent)
                tmpParent->findID(linkID, getOppositeDirection(),
                                  workingList)
                derive newLink from tmpParent to refParent
            tmpList = tmpList->next
        if (workingList == NULL)
            return
        else
            tmpList = workingList
            workingList = NULL
```

Figure 4-10: Pseudo-code for **deriveTransitive** routine. The C++ implementation of this code can be seen in **Appendix B**.

This routine is called on the subepisodes which were created from adding a new ***contains*** link between *episode2* and *episode3*. It traverses up all the existing ***contains*** links and attempts to derive a new link between any new parent that is found to the episode on which the new link terminates.

4.4.3 DeriveSameAs Routine

The final major algorithm that is used for link derivation is the **deriveSameAs** routine. This func-

tion is called whenever a ***sameAs*** link is created between two episodes. Essentially, an "equality" relation is created, and the system attempts to make both episodes look identical in terms of the connections they have with other episodes. **Figure 4-11** shows an example of this kind of scenario. The solid lines represent links that were input into the system. The dotted lines denote derived links.

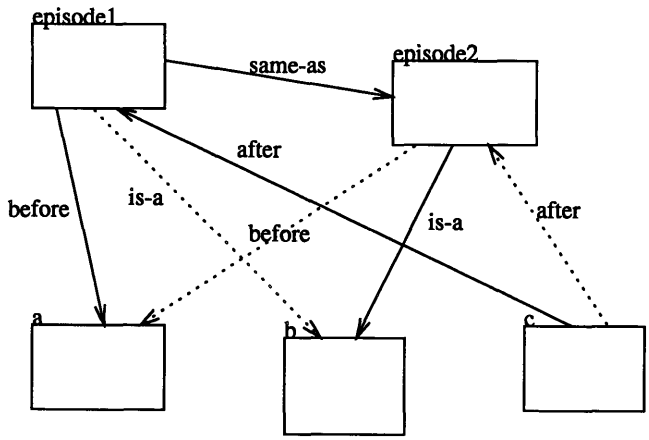

Figure 4-11: Results of derivation when ***sameAs*** relation is created between *episode1* and *episode2*.

The pseudo-code for **deriveSameAs** is shown in **Figure 4-12** below. When this routine is called, both episodes connected by the ***sameAs*** relation will contain the same kind and number of links emanating and terminating between them, and any duplicates will also be disregarded.

Other routines have also been implemented to derive other information. Because these functions, **checkTransitive**, and **checkReflection**, and **deriveNegativeTransitive** all possess algorithms similar to the ones mentioned above, they will not be expanded.

The system described above thus allows users to create many different types of scenarios with the links provided and episodes which they can draw and manipulate on the screen. Rudimentary inconsistency checks have also been provided. To save on resources, these checks will check on inconsistencies among episodes which are interconnected. In addition, deductive inference techniques are also provided to derive new information from the given representation. The interface and implementation functions are all listed in **Appendix A**. A brief description of each function are provided, along with the each function's arguments and return value. **Appendix B** contains the source code for all the member functions.

```
kRepSubEpisode::deriveSameAs(list)
        parent = findParent(list)
        terminationParent = getTermination(list)->findParent(list)
        parent->findID(null, none, tmpList)
        while (tmpList != NULL)
            get current subepisode from tmpList
            find termination of current
            tmpParent = termination->findParent(list)
            if no duplicate link found
                if (termination->getDirection() == destination)
                    derive newLink from termParent to tmpParent
                else
                    derive newLink from tmpParent to termParent
            tmpList = tmpList->next
```

Figure 4-12: Pseudo-code for **deriveSameAs** routine. The C++ implementation of this code can be seen in **Appendix B**.

Chapter 5 Examples from the Visualization System

The system described in **Chapter 4** is implemented in C++ and runs in the Windows environment. It is a tool which allows high-level design and analysis of scenarios with participatory semantics. It allows users to draw boxes and lines on the screen to denote episodes and links which represent objects and their interrelationships in the real world. In this chapter, its features and limitations will be explored. Examples will be presented which demonstrate the deductive inference and consistency checking schemes described in the previous chapter.

Participatory semantics seeks to derive additional relationships and dependencies among participants from what is already present in the knowledge base. The new knowledge is usually derived from rules and properties found in mathematics and logic.

5.1 Transitivity Derivation

One of the most common properties is the transitive property of equality. In **Chapter 4**, the definition from mathematics and its analogical definition in participatory semantics were both presented. Now, the system will use that property on its knowledge base.

In **Figure 5-1**, the following representation from the Museum of Science is entered into the knowledge base. Lexically, this representation is seen as

***contains** (3 West, Computers)*
***contains** (3 West, Birds)*

Suppose then that a user adds onto this representation the fact that

***contains** (3rd Floor, 3 West)*

Through transitivity, two more interconnections should be added to represent the fact that

***contains** (3rd Floor, Computers)*
***contains** (3rd Floor, Birds)*

The result of this calculation can be seen in **Figure 5-2**. Links have been created between the related episodes; thus, the knowledge base has been updated. The direction of the link is evident by the small squares

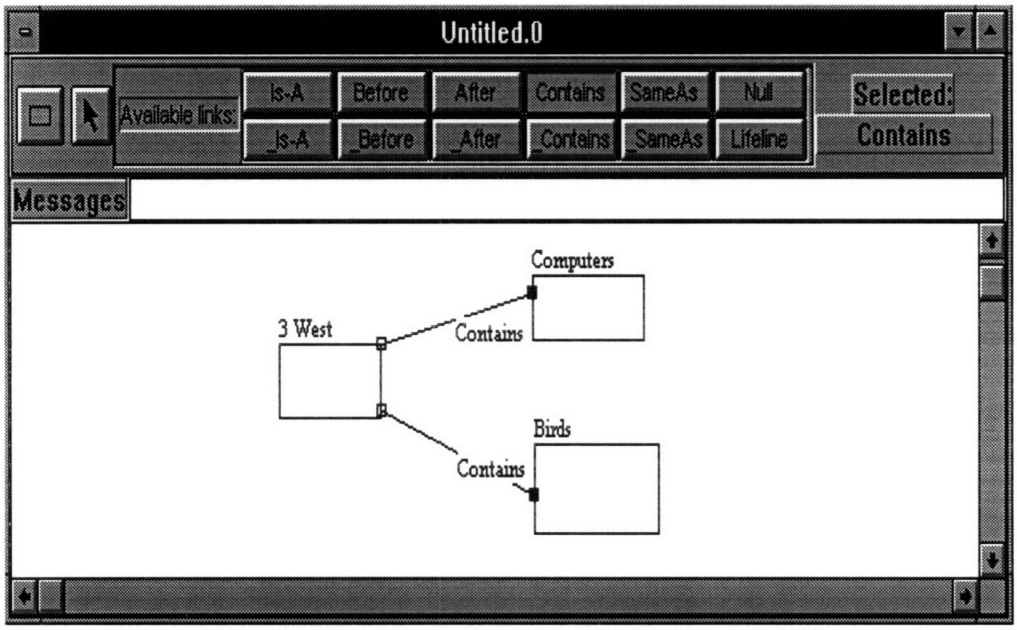

Figure 5-1: Two representations were added to this system and can be seen above:
contains *(3 West, Computers)*
contains *(3 West, Birds)*

that symbolize the subepisodes in the episode. Unfilled ones denote the origination of the link while the filled ones denote the destination.

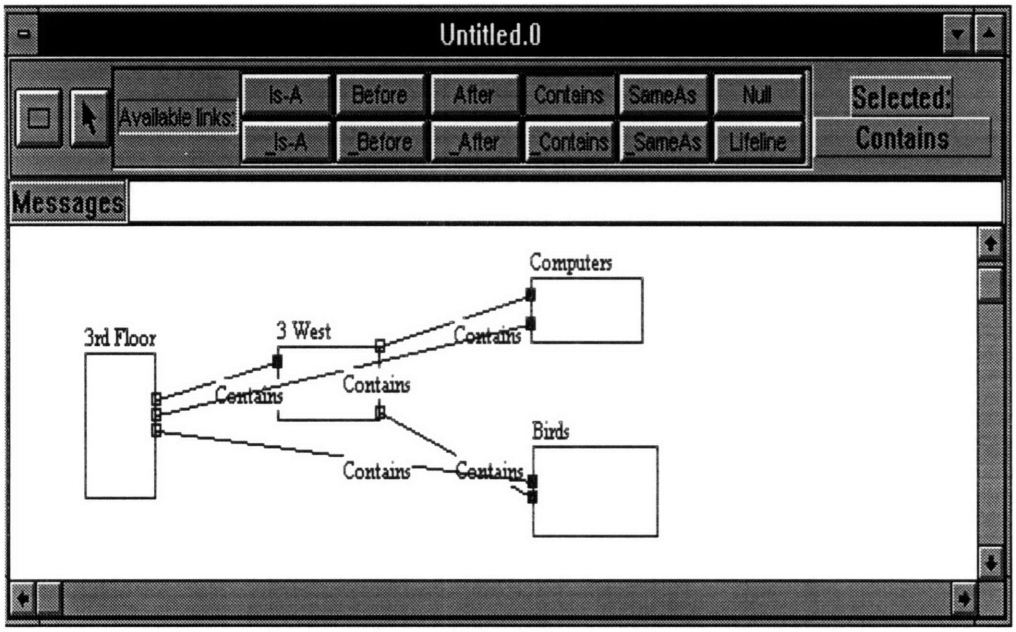

Figure 5-2: To illustrate transitivity, one more relation has been added to **5-1**:
contains *(3rd Floor, 3 West)*
Adding the above relation creates two more relations:
contains *(3rd Floor, Computers)*
contains *(3rd Floor, Birds)*

5.2 Negative Transitivity Derivation

Now suppose that the user had inputted the following instead

~contains *(3rd Floor, 3 West)*

The tilde in front of ***contains*** denotes the negation of this link. With the addition of this link, the system will call on the **deriveNegativeTransitive** property, and the following new representations will be added to the knowledge base

~contains *(3rd Floor, Computers)*
~contains *(3rd Floor, Birds)*

The result of this calculation can also be seen in **Figure 5-3**.

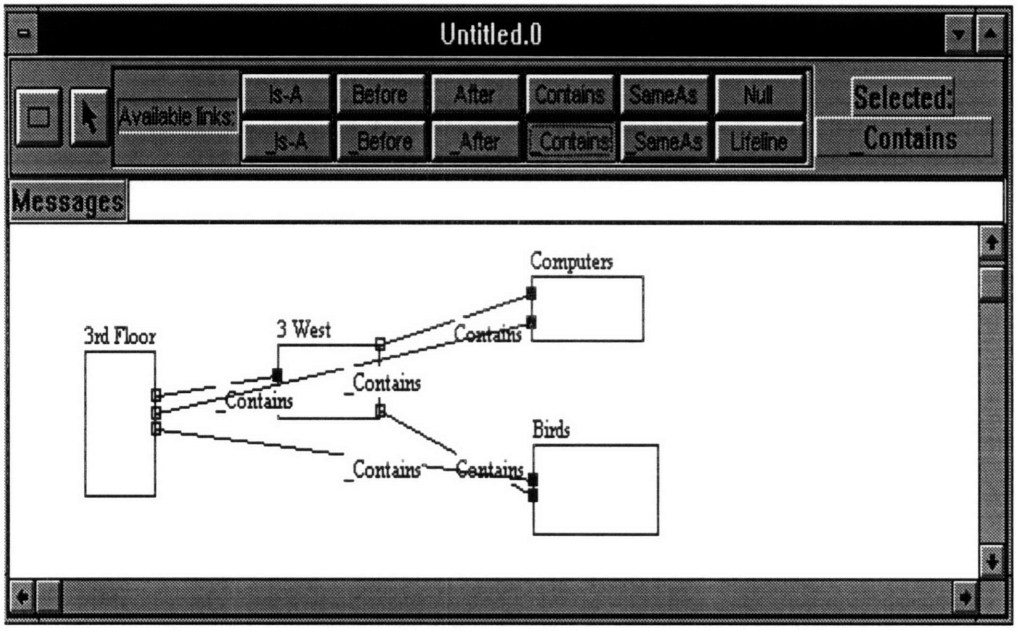

Figure 5-3: To illustrate negativity transitivity, one more relation has been added to **5-1**:
~contains *(3rd Floor, 3 West)*
Adding the above relation creates two more relations:
~contains *(3rd Floor, Computers)*
~contains *(3rd Floor, Birds)*

5.3 SameAs Derivation

In addition to transitivity, the previous chapter also discussed deriving new relations from the **sameAs** link. Using **Figure 5-1** again, suppose another episode is created called *West 3*. This can be an episode that becomes a private, personal copy in the user's local knowledge-base. If a ***sameAs*** link is established

sameAs *(West 3, 3 West)*

then the following relations should also be derived

contains *(West 3, Computers)*
contains *(West 3, Birds)*

Figure 5-4 shows how the system responds to the additional knowledge.

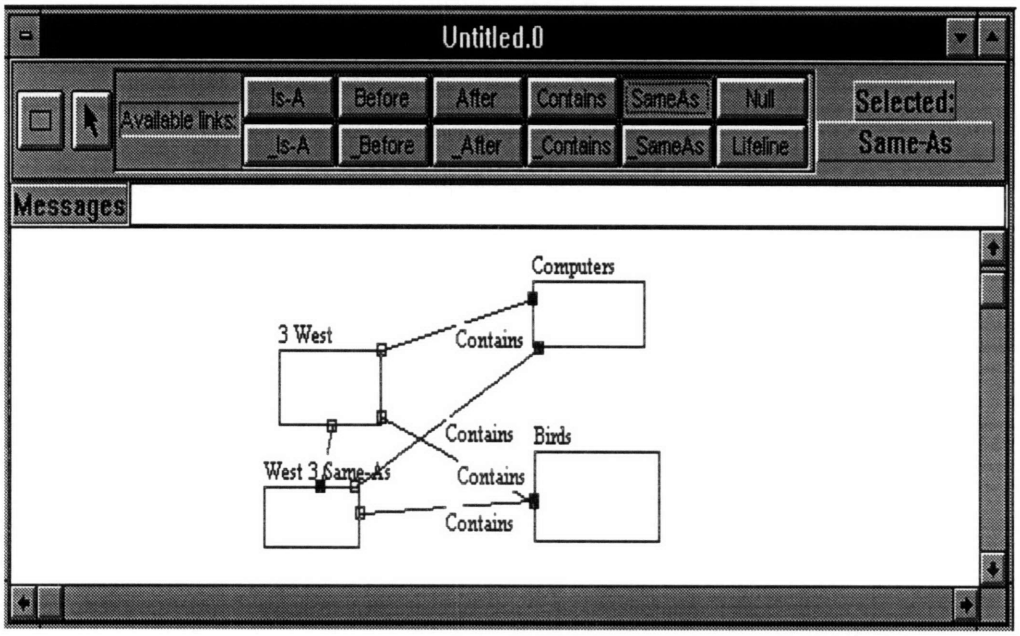

Figure 5-4: To illustrate the ***sameAs*** relation, one more relation has been added to **5-1**:
sameAs *(West 3, 3 West)*
Adding the above relation creates two more relations:
contains *(West 3, Computers)*
contains *(West 3, Birds)*

Both episodes that are connected by the ***sameAs*** link have the same connections to the same episodes.

5.4 Internal Reflection Check

In addition to knowledge derivation, this system also provides rudimentary consistency checks. These checks iterate up all the interrelationships to which an episode is linked. By just looking at established links, the system will not become inundated with unnecessary calculations by checking unrelated episodes. An inconsistency in representation will force a message to be created and sent to the user. In the present implementation, negotiation is only used to detect and not resolve inconsistencies. Negotiation will only be used when conflicts arise which block other action from taking place. TeleVisit will thus use negotiation for its own conflict resolution.

Temporal relationships will be used to talk about some of the inconsistencies that can be encountered. Suppose that the following representations are input to the system.

before *(event1, event2)*

before *(event2, event3)*
before *(event3, event1)*

The last relationship will cause an inconsistency, and the message will be broadcast back to the user because it is creating an internal reflection that is not valid for a ***before*** link. This example can be seen in **Figure 5-5**.

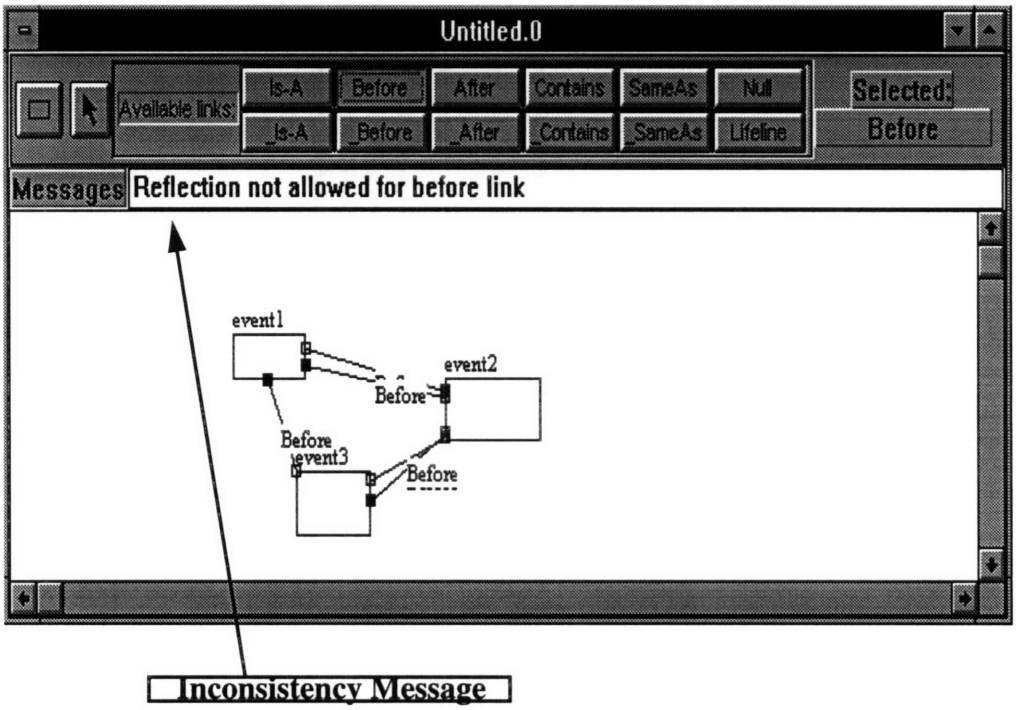

Figure 5-5: Internal reflection inconsistency caused by these three relationships:
before *(event1, event2)*
before *(event2, event3)*
before *(event3, event1)*
Notice the message in the box pointed to by the arrow.

Notice the message that appears in the **Messages** box. Notice also that additional links have been derived from even this inconsistent link. This is because although an inconsistency has been detected, there is no knowledge as to which relationship is indeed correct, and no negotiation is being used. So assumptions have been made that all links entered by the user are correct. For the rest of the examples in this chapter, this type of behavior will always be true.

5.5 Link-Negation Check

Another inconsistency may arise if a link and its negation are both created between two episodes.

before *(event1, event2)*
~before *(event1, event2)*

Figure 5-6 gives an example of this and its corresponding error message.

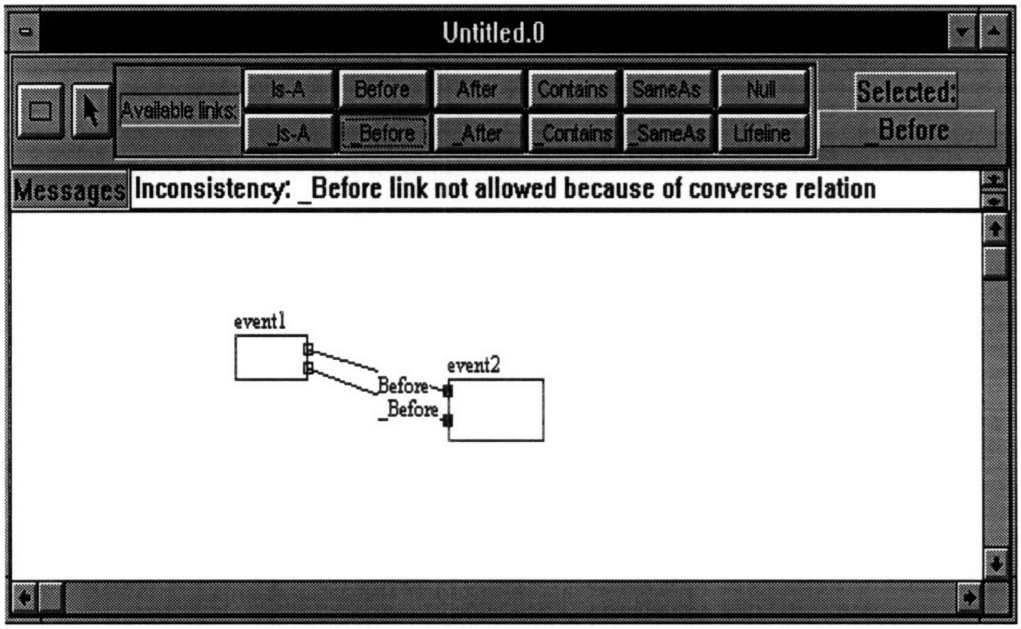

Figure 5-6: Link-negation inconsistency caused by these two relationships:
before *(event1, event2)*
~before *(event1, event2)*

5.6 Link-Reflection Check

Finally, another common inconsistency arises because the reflexive property simply does not hold for some links. The following pairs of representations cannot coexist together.

before *(event1, event2)*	***after*** *(event1, event2)*	***contains*** *(a, b)*
before *(event2, event1)*	***after*** *(event2, event1)*	***contains*** *(b, a)*

These inconsistencies are also checked for in the system and are brought to the attention of the user. **Figure 5-7** demonstrates how the system handles this inconsistency along with the error message it sends.

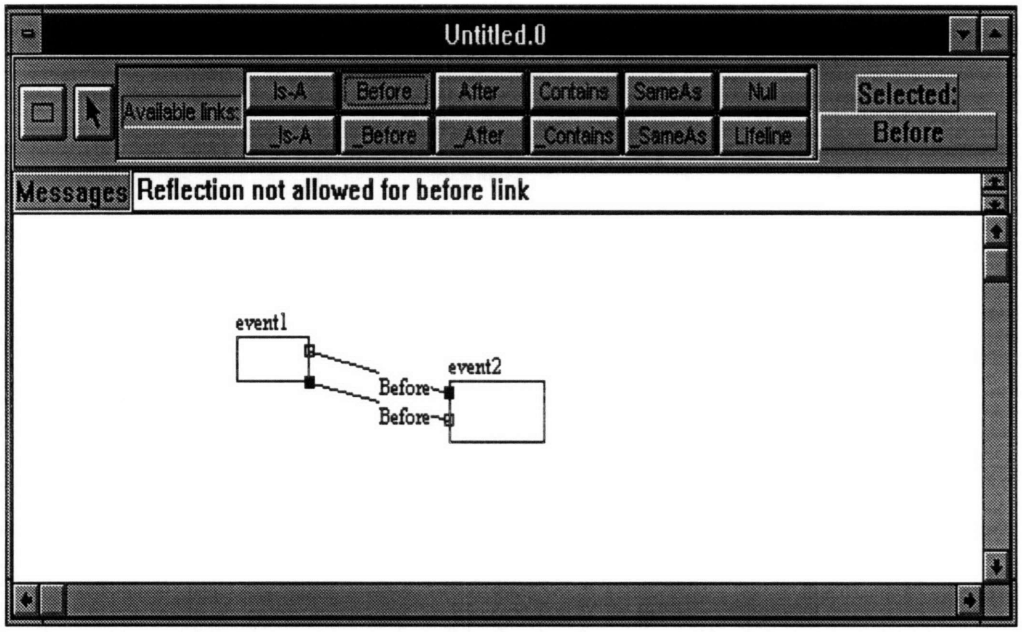

Figure 5-7: Link-reflection inconsistency caused by these two relationships:
before *(event1, event2)*
before *(event2, event1)*

Along with knowledge derivation and inconsistency checking, this system contains many features which allow easier modification and creation of scenarios. When an episode is deleted, then any episode that it created will also be destroyed. Creation of an episode will cause any existing episodes which is encompassed by this one to be added to its subepisode list. Conversely, any episode which encompasses this one will also add this to its subepisode list. Clicking on an episode will make it the current one and allow it to be deleted. And, finally, double-clicking on an episode will allow it to be edited--its name can be changed and its subepisodes viewed. Handling of these mouse events provide for a good system for handling the graphics required of participatory semantics.

Chapter 6 Conclusion

6.1 What has been Covered

A system has been built for the visual analysis and construction of scenarios based on participatory semantics. Participatory semantics technology will be the backbone behind the Continually Available Telecomputing Systems Infrastructure (CATSI). TeleVisit, a project for the Museum of Science which will incorporate CATSI, will be the synergy that binds solutions to issues prevalent in software engineering to similar issues prevalent in multi-user knowledge base systems. Many issues in knowledge representation technology regarding non-monolithic, multi-model and multi-user systems are treated in participatory semantics. These issues have strong correlations with the problems that large-scale software systems are encountering now. Just as software systems are required to combine different architectures, data formats, and program specifications, participatory semantics systems must be able to work with knowledge bases on local machines whose owners all have different perspective of the world, different ideas as to how the knowledge base works, and different levels of expertise with the technology.

Mediation is the key idea which the system will use to interpret the various information that users input from different machines. It will use the underlying methods and rules for participation that help to determine the meanings of various representations. Negotiation is the key for resolving the many problems in participatory semantic. With negotiation, participants can make representations about conflicts, inconsistencies, and timing issues which may arise. Thus using mediation and negotiation, a CATSI system will seek to find the meanings for user inputs and act as the negotiator for participants who are having conflicts.

The system that was built for this thesis has its roots in participatory semantics. It allows users to make their own representations about their perception of the world. Objects in the world become episodes, and the relationships between those objects become links. Various links have different properties which allow different relationships to be derived. For example, many links display transitivity and others disallow reflexivity. These properties are very much ingrained in mathematics and the technique used to derive these properties is called deductive inference.

Relationships between episodes are derived and their consistency with the rest of the knowledge

base is partially checked on every time a link is created between two episodes. Only the episodes which are directly and nondirectly linked to the episodes being operated upon are included in the calculations. With this scheme, resource use is kept at a minimum because VOPS will not check for inconsistencies with other non-related scenarios.

The intention of this system is to provide a background for TeleVisit, a project for the Museum of Science. The algorithms, objects, and their interfaces used will be built upon by TeleVisit. When the user requests a service from TeleVisit, it will automatically create the semantics to represent the person's activity. For example, TeleVisit will be able to track a user's movements in the museum. Portable computers carried by a user will be equipped with a navigation system which will broadcast and receive signals from location sensors placed around the museum. When the user enters a part of the museum, TeleVisit may create a relationship such as

***contains** (3 West, user1)*

to represent the fact that *user1* is currently in *3 West.*

TeleVisit will also be able to help users arrange a visit using participatory semantic. Suppose the user would like to see a series of scheduled events. She may enter the following scenario into TeleVisit by pulling objects from the database and drawing relationships between them.

***before** (Omni Theater, planetarium)*
***before** (planetarium, reptile demo)*
***before** (reptile demo, 12PM)*

The system can thus create more representations between events and their respective schedules. Internally, it will search events to avoid conflicts and allow the user to go to every event before noon.

Finally, **Chapter 3** gave an example of finding a lost child. How a scenario like this can be resolved can be programmed into TeleVisit. Other situations are all possible. These can be scenarios such as emergencies--what to do in case of a fire; conflicting interests--John wants to see the theater of electricity, but Jane wants to see the Omni Theater show; and random queries--I'd like to know where group1 went before going to see the reptile demonstration.

Participatory semantics will be the backbone for CATSI and TeleVisit. Its suitability as a semantics is very evident in the five roles of a knowledge representation that it satisfies [Davis et al 93]. These five

roles which [Davis et al 94] enumerated provide a useful understanding of a representation for both end users and developers who will be extending and improving this technology. The first role is that it should be used as a surrogate for the real world. Objects from the real world, whether real or conceptual, formal or informal, can all be modelled as episodes in this representation, and the relationships between all of these episodes can all be modelled as links. This provides a very clear and obvious way of bringing the world into perspective.

The second role for participatory semantics is that it should be a set of ontological commitments. This semantics thus focuses on modelling the spatial and temporal relationships between objects. It does not purport to find order within the chaos of the universe, nor does it care. The third role which participatory semantics claims to have filled is that it is a fragmentary theory of intelligent reasoning. Mathematical concepts have been used as the formal methods in participatory semantic to check for consistency and derive additional knowledge from the knowledge base.

Fourthly, participatory semantic is a medium for efficient computation. C++ constructs will be heavily used for creation of objects and derivation of subclasses. The present implementation of the episode-class objects allows for easy traversal down episodes connected by links. Traversing down linked episodes will be as easy as calling interface functions and dereferencing pointers in C++. Finally, this representation technology has filled the fifth role as a medium of human expression. Simple sentences such as

The exhibit is on the 3rd floor

can be translated lexically into

***contains** (3rd Floor, exhibit)*

and more graphically into **Figure 6-1**.

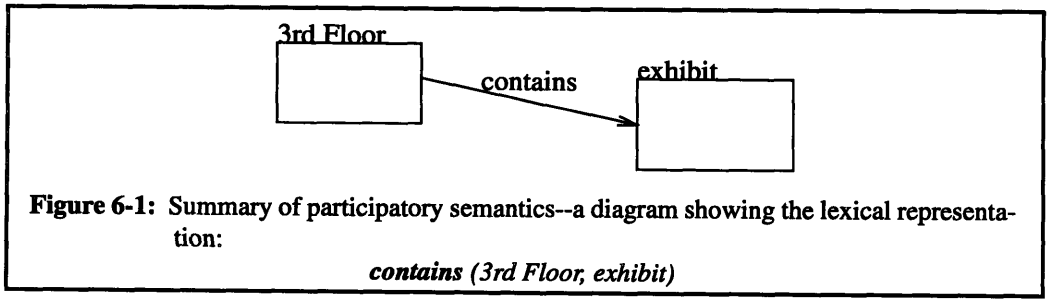

Figure 6-1: Summary of participatory semantics--a diagram showing the lexical representation:

***contains** (3rd Floor, exhibit)*

The most important strength of participatory semantics lies in its strategy of using negotiation to solve common issues present in multi-user and continually available systems. Conflicts, inconsistencies, and timing issues are all big problems which many systems face. Participatory semantic will negotiate to enable participants in the system to make further representations about themselves and their knowledge of the outside world in order to resolve issues. As seen by some running examples of the visualization system, this has indeed been possible on a relatively small scale.

The weaknesses of this VOPS are ones that plague many large-scale software systems. Inconsistency is a big issue. How should it be tackled? Should it be dealt with at all? These are all questions which will affect performance and may affect the integrity of the knowledge base. In addition, the heterogeneous user base may also hamper the usefulness of the system. TeleVisit will be a good buffer which will keep all the semantics in the background and present an interface which hides all the implementation details of the semantics. However, the "user" may also be other developers who need to derive more classes from those given and provide richer functionalities to the average museum-goer. These people will indeed have different perceptions on the roles of the knowledge representation technology. Thus, they must be educated on the five roles which participatory semantics seeks to fulfill.

6.2 Extensions for Further Research

The work presented here presents a general framework for which later work can follow. The visualization system will be used to test the strengths and weaknesses of participatory semantics. However, it requires much refinement. More work needs to be done perfecting the class structure that was created. For instance, in the present implementation of VOPS, the subepisodes described only reference one other subepisode. This implies that subepisodes can only allow unidirectional links. However, there may be subepisodes which reference two or more subepisodes. These will perhaps allow more versatile links which can specify not just binaries relationships such as

***link** (episodeA, episodeB)*

but also maybe multi-episode relationships which allow

***link** (episodeA, episodeB, episodeC, ...)*

The types of links which will be available to the user will definitely need to be expanded. For TeleVisit, these links have to describe relationships which may appear within the Museum of Science. For CATSI, as many links as possible that will describe the relationships between objects in the world have to be created. Besides temporal and spatial links, there may also be links which describe an organizational hierarchy which outlines the structure of a company; a priority list which weighs certain tasks based on importance; or maybe a decision tree which can make choices that are dependent on outside forces.

In addition, the system needs to be enhanced to store its semantics in a persistent object base. All classes have to be persistent, and their instances need to be stored in the database and allowed to be accessed at will. This database will act as a repository for predefined episodes and links. Users just need to use those objects to create their knowledge base on their own local machines. There will be a taxonomy of all the possible episodes and links inside the database, and there need to be ways of maintaining and expanding those episodes and links.

TeleVisit, which will be the first system to use participatory semantics, can derive from the episode class objects such as exhibits, events, and schedules. Relationships between them can then also be immediately derived from the predefined links provided in the system or they can be derived from the base link class. The aim of TeleVisit, however, is not to allow end users (museum-goers) to create their own semantics of their visit, but to be used as a tool that people can use inside the museum to perform various services. In addition, TeleVisit should be an exhibit in and of itself that resides within the museum. The underlying semantics will all be created within the system as the person makes demands on the system.

Furthermore, TeleVisit will be used to test the feasibility of negotiations in resolving conflicts. Types of conflicts have to be identified, categorized, and dealt with perhaps on a case by case basis. The system will need to know what kind of further semantics are needed to resolve a scheduling conflict or, as in **Chapter 3,** find a lost child. Much work needs to be done for those contingencies so that certain methods and rules can be used to perform the negotiation.

Finally, algorithms were presented in this paper which could detect inconsistencies in semantics and also deduce additional knowledge. These performed their calculations based only on episodes which were interconnected. Other more thorough mechanisms for performing those tasks need to be investigated

that can also derive knowledge from unconnected episodes. Consistency checking, however, may not be a priority, and the scheme used in VOPS is a compromise between checking the entire database and not checking at all.

A system for analyzing and visualizing participatory semantics technology has thus been built. Named VOPS, this system will be a precursor for the TeleVisit system and the Continually Available Telecomputing Infrastructure. The strengths of this system have been discussed and the limitations of such a representation have also been presented and elaborated. Hopefully, the fruits of this labor will be seen when implementation of the TeleVisit project begins.

Appendix A Interface Specifications for Episode and Subclasses of Episode

kRepEpisode

Summary

kRepEpisode is the base class for participatory representation. It is a container object for other episodes. It will keep track of all of the episodes contained within it.

Hierarchy

kRepEpisode is a base class

Member Functions

Constructors

kRepEpisode

kRepEpisode(void);

Summary

Default constructor.

kRepEpisode(const kRepEpisode &*orig*);

orig Reference to original kRepEpisode object

Summary

Copy constructor.

kRepEpisode(const kRepEpisode &*orig*, const EosRect &*frame*);

orig Reference to original kRepEpisode object

frame Reference to a frame in which to draw orig

Summary

Creates a new kRepEpisode with the desired *frame*.

Destructor

~kRepEpisode

~kRepEpisode(void);

Summary

Default destructor.

Operator Overloads

operator =

kRepEpisode &**operator=**(const kRepEpisode &*orig*);

orig Reference to original kRepEpisode object

Return
this
Summary
Default assignment operator

Member Access Functions

addEpisode

virtual void ***addEpisode***(kRepEpisode **ep*);
ep Pointer to an episode
Summary
Adds *ep* to the *episodeList* of this episode.

virtual void ***addEpisode***(kRepEpisode **ep*, int *i*);
ep Pointer to an episode
i integer value
Summary
Adds *ep* to the *episodeList* at the location specified by *i*.

edit

virtual void ***edit***(kRepView &*view*);
view Reference to the view which holds the episode.
Summary
Pops up a dialog which allows editing of many of the member data in a kRepEpisode.

getEpisodeList

kRepEpisodeList ****getEpisodeList***(void);
Summary
Returns a pointer to the *episodeList* contained in this object.

getCreator

kRepEpisode ****getCreator***(void);
Return
Pointer to the kRepEpisode object which created this one.

getName

EosString ***getName***(void);
Return
An EosString object that is the name of the episode.

parentIsDeletable

void ***parentIsDeletable***(EosBoolean *bool*);
bool EosBoolean object which the parentDeletable flag is set to.

removeEpisode

virtual EosBoolean ***removeEpisode***(kRepEpisode **ep*);

ep Pointer to a kRepEpisode object

Return

EosBoolean specifying whether the function did find the episode, *ep*, for deletion.

selfControllable

void ***selfControllable***(EosBoolean *bool*);

bool EosBoolean object which the selfControl flag is set to.

EosBoolean ***selfControllable***(void);

Return

EosBoolean specifying the value of the *selfControl* flag.

Screen Interfaces

addContainedEpisodes

virtual void ***addContainedEpisodes***(kRepEpisodeList **lst*)

lst Pointer to the list of episodes in a scenario

Summary

Adds all episodes from *lst* which reside physically within this episode to its *episodeList*.

findContainerEpisode

kRepEpisode ****findContainerEpisode***(kRepEpisodeList **lst*);

lst Pointer to the list of episodes in a scenario

Summary

Finds the episode from *lst* which physically owns this episode.

containsEpisode

virtual EosBoolean ***containsEpisode***(kRepEpisode **ep*);

ep Pointer to the episode being tested

Return

EosBoolean specifying whether the result is true

Summary

This interface checks to see if *ep* is indeed a member of the eipsode's *episodeList*.

containsPoint

virtual EosBoolean ***containsPoint***(const EosPoint &*pt*);

pt Reference to the point to be tested

Return

EosBoolean specifying whether the result is true

Summary

This interface checks to see if *pt* lies within this episode.

draw

void ***draw***(kRepView &*view*);

view Reference to the window in which to draw

Summary

Draws the episode in the window specified by *view*.

drawFeedback

void drawFeedback(kRepView &*view*);

view Reference to the window in which to draw

Summary

Same as draw except reshaping of the episode can be seen.

Interfaces with Other Episodes

allowSubEpisode

virtual EosBoolean ***allowSubEpisode***(kRepSubEpisode **ep*,
kRepEpisodeList **lst*);

ep pointer to a subepisode to be tested for.

lst pointer to a list of all episodes in a scenario.

Return

EosBoolean which determines whether this episode, *ep*, is allowed in the scenario specified by *lst*.

findID

virtual int ***findID***(tools *t*, direction *d*, kRepEpisodeList **lst*);

t enumeration to the type of link searching for

d enumeration to the direction of the link

lst pointer to an episode list which will store the result

Return

int specifying how many were found

Summary

The *episodeList* in the episode is searched to find all subepisodes which match the criteria *t* and *d*. Every subepisode found will be added to *lst*, and the number of subepisodes found will be returned.

findParent

kRepEpisode ****findParent***(kRepEpisodeList **lst*);

lst list of all the episodes in a scenario.

Summary

Given a list of episodes, this interface will attempt to find the episode that owns this one. If there are none, then it returns NULL.

isParentOf

virtual EosBoolean ***isParentOf***(kRepEpisode **ep*);

ep pointer to a kRepEpisode object

Return

EosBoolean specifying whether this episode is indeed the parent of *ep*.

removeAssociations

virtual void ***removeAssociations***(kRepEpisodeList **lst*, EosBoolean *bool*);

lst list of all the episodes in a scenario.

bool EosBoolean specifying whether the memory should be destroyed.

Summary

Given a list of episodes, this interface will seek to remove all references to it from all episodes. If *bool* is true, then the memory which holds this object will be destroyed.

kRepLink

Summary

kRepLink is the base class for the connections between episodes.

Hierarchy

kRepEpisode

kRepLink

Member Functions

Constructors

kRepLink

kRepLink(void);

Default constructor.

kRepLink(const kRepLink &*orig*);

orig Reference to original kRepLink object

Summary

Copy constructor.

kRepLink(const EosRect &*frame*);

frame Reference to a frame in which to draw the link

Summary

Creates a new kRepLink with the desired *frame*.

kRepLink(const kRepLink &*orig*, const EosRect &*frame*);

orig Reference to original kRepLink object

frame Reference to a frame in which to draw orig

Summary

Creates a new kRepLink with the desired *frame*.

Destructors

~kRepLink

~kRepLink(void);

Summary

Default destructor.

Operator Overloads

operator =

kRepLink ***&operator=***(const kRepLink *&orig*);

orig Reference to original kRepLink object

Return

**this*

Summary

Default assignment operator

Member Access Functions

setFromEpisode

virtual void ***setFromEpisode***(kRepEpisode **ep*);

ep Pointer to a kRepEpisode

Summary

This just sets the location in the *episodeList* of the link to *ep*.

setToEpisode

virtual void ***setToEpisode***(kRepEpisode **ep*);

ep Pointer to a kRepEpisode

Summary

This just sets the location in the *episodeList* of the link to *ep*.

getFromEpisode

virtual kRepEpisode ****getFromEpisode***(void);

Return

A pointer to a kRepEpisode object which is the origination of the link.

getToEpisode

virtual kRepEpisode ****getToEpisode***(void);

Return

A pointer to a kRepEpisode object which is the destination of the link.

Screen Interfaces

containsEpisode

virtual EosBoolean ***containsEpisode***(kRepEpisode **ep*);

ep Pointer to the episode being tested

Return

EosBoolean specifying whether the result is true

Summary

This interface will always return false because links are denoted by a line. So, lines cannot contain anything.

containsPoint

virtual EosBoolean ***containsPoint***(const EosPoint &*pt*);

pt Reference to the point to be tested

Return

EosBoolean specifying whether the result is true

Summary

This interface checks to see if *pt* lies on the link.

draw

void ***draw***(kRepView &***view***);

view Reference to the window in which to draw

Summary

Draws the episode in the window specified by *view.*

drawFeedback

void ***drawFeedback***(kRepView &*view*);

view Reference to the window in which to draw

Summary

Same as draw except reshaping of the episode can be seen.

Interfaces with Other Episodes

setupLink

virtual void ***setupLink***(kRepSubEpisode **from*, kRepSubEpisode **to*, tools *t*, kRepEpisodeList **lst*);

from Origination of the link

to Destination of the link

t Type of link

lst List of links which make up the scenario

Summary

This interface attempts to setup a link from two already established subepisodes which act as anchors. The type of link is specified by *t*, and the present scenario is in *lst.*

derive

virtual void ***derive***(kRepEpisode **from*, kRepEpisode **to*, tools *t*, kRepEpisodeList **lst*);

from Origination of the link

to Destination of the link
t Type of link
lst List of links which make up the scenario
Summary
This interface attempts to derive a link from two episodes. The type of link is specified by *t*, and the present scenario is in *lst*. Necessary subepisodes are all created within this interface.

kRepSubEpisode

Summary

The kRepSubEpisode class provides points for anchoring links to episodes. Subepisodes contain all the logic that are used to derive links and check for inconsistencies. In the base implementation, subepisodes only reference one other subepisode.

Hierarchy

kRepEpisode
kRepSubEpisode

Member Functions

Constructors

kRepSubEpisode

kRepSubEpisode(void);
Default constructor.

kRepSubEpisode(const kRepSubEpisode &*orig*);
orig Reference to original kRepSubEpisode object
Summary
Copy constructor.

kRepSubEpisode(const EosRect &*frame*, tools *t*, EosBoolean *bool*, direction *d*, EosString *str*, kRepEpisode **ep*);
frame Reference to desired frame
t Type of link which this subepisode anchors
bool EosBoolean to specify parentDeletable
d Direction which subepisode signifies
str EosString object
ep Pointer to a kRepEpisode object
Summary
Constructor which creates a subepisode with given frame, linkID *t*, and direction *d*. The *parentDeletable* flag is set to *bool*, *name* is set to *str*, and *creatorEpisode* is set to *ep*.

kRepSubEpisode(tools *t*, EosBoolean *bool*, direction *d*, EosString *str*);

t Type of link which this subepisode anchors

bool EosBoolean to specify parentDeletable

d Direction which subepisode signifies

str EosString object

Summary

Constructor which creates a subepisode with given linkID *t* and direction *d*. The *parentDeletable* flag is set to *bool* and *name* is set to *str*.

Destructors

~kRepSubEpisode

~kRepSubEpisode(void);

Summary

Default destructor.

Operator Overloads

operator =

kRepSubEpisode ***&operator=***(const kRepSubEpisode *&orig*);

orig Reference to original kRepSubEpisode object

Return

**this*

Summary

Default assignment operator

Member Access Functions

alwaysGetNegativeLink

tools ***alwaysGetNegativeLink***(void);

Return

Enumeration which always specifies a negative link which the subepisode is anchoring.

getDirection

direction ***getDirection***(void);

Return

Enumeration which specifies the direction of the link which the subepisode is anchoring.

getOppositeDirection

direction ***getOppositeDirection***(void);

Return

Enumeration which specifies the opposite direction of the link which the subepisode is anchoring.

getLinkID

tools ***getLinkID***(void);

Return

Enumeration which specifies the type of link which the subepisode is anchoring.

getConverseLink

tools ***getConverseLink***(void);

Return

Enumeration which specifies the negation of the link which the subepisode is anchoring.

Interfaces with Other Episodes

allowSubEpisode

EosBoolean ***allowSubEpisode***(kRepSubEpisode **ep*, kRepEpisodeList **lst*);

ep Pointer to the subepisode being checked

lst Pointer to list of episodes in scenario

Return

EosBoolean specifying whether the new subepisode will cause an inconsistency with this one.

Summary

This routine checks many rules and calls the correct implementations procedures to see if *ep* is allowable with this subepisode. See **Appendix B** for the implementations.

checkDependence

EosBoolean ***checkDependence***(kRepEpisodeList **lst*, kRepSubEpisode **ep*, tools *t*, direction *d*);

lst Pointer to list of episodes in scenario

ep Pointer to a subepisode

t Enumeration for type of link that subepisode is holding

d Enumeration for direction of link

Return

EosBoolean specifying whether there is dependence between this subepisode and *ep*.

Summary

This interface takes a scenario, a subepisode being checked for, the link type that this subepisode is dependent on, and the direction of the link. It checks to see if this subepisode is logically dependent to ep.

deriveNew

virtual void ***deriveNew***(kRepEpisodeList **lst*);

lst Pointer to list of episodes in the present scenario

Summary

Attempts to derive additional knowledge from the scenario specified in *lst*. This interface calls on the following implementations: ***deriveTransitive***, ***deriveNegativeTransitive***, ***deriveSameAs***.

findDuplicate

EosBoolean ***findDuplicate***(kRepEpisode **ep*, kRepEpisodeList **lst*);

ep Pointer to the episode which holds the destination of the link

lst Pointer to list of episodes in the present scenario

Return

EosBoolean if a duplicate link is found.

Summary

This interface will return true if it finds that there is already a link of the same type as this subepisode's link type and going in the same direction, else it will return false.

getReferenced

kRepSubEpisode ***getReferenced***(void);

Return

Pointer to subepisode that is referenced to by this one.

getTermination

kRepSubEpisode ***getTermination***(kRepEpisodeList **lst*);

lst List of episodes in the present scenario

Return

Pointer to subepisode that is at the other end of the link which this one is anchoring.

initLink

virtual void ***initLink***(kRepLink ***link*, tools *t*);

link Pointer to a pointer to a kRepLink

t Enumeration to type of link

Summary

This interface will create a *link* of type *t*.

Appendix B Source Code Listings

This appendix contains source code listings for all the header files and functional definitions for all the classes used in the project. The files are listed in alphabetical order in the following order

.cpp	Member function definitions
.hpp	Main header file
.pre	Class pre-definition header file
.non	Non-Interactive header file (i.e., background definitions and member data)

```
// after.cpp
// User file generated by Utah.
// All user changes should be done in this file.
// This file is generated only once, it won't be regenerated.
// The full skeleton is available in after.csk.

#ifndef AFTER_HPP
#include <after.hpp>
#endif

#ifndef PROBECOM_HPP
#include <probecom.hpp>
#endif

afterLink::afterLink() : kRepLink()
{
  name="After";
}

afterLink::~afterLink()
{
}

afterLink &afterLink::operator =(const afterLink &orig)
{
 EOS_PROBE_LOCK(*this);
 if (this != &orig)
 {
  kRepLink::operator=(orig);
 }
 return *this;
}

afterLink::afterLink(const afterLink &orig) : kRepLink(orig)
{
}

afterLink::afterLink(const kRepLink &orig, const EosRect &frame) : kRepLink(orig)
{
  name="After";
```

```
  shape->setFrame(frame);
}

void afterLink::setupLink(kRepSubEpisode *from, kRepSubEpisode *to, tools id, kRepEpisodeList *lst)
{
  kRepLink::setupLink(from, to, id, lst);
}
```

```
// after.hpp
// Header Generated by Utah, don't tread on me.
// Portions Copyright 1994 ViewSoft, Inc. All rights reserved.
#ifdef EOS_STRICT_INCLUDE
 #ifdef AFTER_HPP
  #error file already included
 #endif
#endif

#ifndef AFTER_HPP
#define AFTER_HPP
// User code for this class is in after
#ifndef EOSDEFS_HPP
#include <eosdefs.hpp>
#endif

#ifndef POINTER_HPP
#include <pointer.hpp>
#endif


///// User Pre-Declaration Code /////
#include "after.pre"

///// End User Pre-Declaration Code /////

#ifndef afterLink_POINTER_DEFINED
#define afterLink_POINTER_DEFINED
EOS_PTR_CLASS(afterLink)
#endif

#ifndef KRLINK_HPP
#include <krlink.hpp>
#endif

class EOS_MODIFIER afterLink: public kRepLink {
public:

public:
 EOS_GET_SDO(afterLink)
 afterLink();
 afterLink(const afterLink &orig);
 virtual ~afterLink();
 afterLink &operator =(const afterLink &orig);
```

```
// Begin Non-Interactive User Class Declaration
public:
#include "after.non"

// End Non-Interactive User Class Declaration
};

#endif // AFTER_HPP
```

// after.pre

// after.non

```
afterLink(const kRepLink&, const EosRect&);
virtual void setupLink(kRepSubEpisode*, kRepSubEpisode*, tools, kRepEpisodeList*);
```

```
// before.cpp
// User file generated by Utah.
// All user changes should be done in this file.
// This file is generated only once, it won't be regenerated.
// The full skeleton is available in before.csk.

#ifndef BEFORE_HPP
#include <before.hpp>
#endif

#ifndef PROBECOM_HPP
#include <probecom.hpp>
#endif

beforeLink::beforeLink() : kRepLink()
{
  name="Before";
}

beforeLink::~beforeLink()
{
}

beforeLink &beforeLink::operator =(const beforeLink &orig)
{
 EOS_PROBE_LOCK(*this);
 if (this != &orig)
 {
  kRepLink::operator=(orig);
 }
 return *this;
}
```

```
beforeLink::beforeLink(const beforeLink &orig) : kRepLink(orig)
{
}

beforeLink::beforeLink(const kRepLink &orig, const EosRect &frame) : kRepLink(orig)
{
  name="Before";
  shape->setFrame(frame);
}

void beforeLink::setupLink(kRepSubEpisode *from, kRepSubEpisode *to, tools id, kRepEpisodeList *lst)
{
  kRepLink::setupLink(from, to, id, lst);
}
```

```
// before.hpp
// Header Generated by Utah, don't tread on me.
// Portions Copyright 1994 ViewSoft, Inc. All rights reserved.
#ifdef EOS_STRICT_INCLUDE
 #ifdef BEFORE_HPP
  #error file already included
 #endif
#endif

#ifndef BEFORE_HPP
#define BEFORE_HPP
// User code for this class is in before
#ifndef EOSDEFS_HPP
#include <eosdefs.hpp>
#endif

#ifndef POINTER_HPP
#include <pointer.hpp>
#endif


///// User Pre-Declaration Code /////
#include "before.pre"

///// End User Pre-Declaration Code /////

#ifndef beforeLink_POINTER_DEFINED
#define beforeLink_POINTER_DEFINED
EOS_PTR_CLASS(beforeLink)
#endif

#ifndef KRLINK_HPP
#include <krlink.hpp>
#endif

class EOS_MODIFIER beforeLink: public kRepLink {
```

```
public:

public:
 EOS_GET_SDO(beforeLink)
 beforeLink();
 beforeLink(const beforeLink &orig);
 virtual ~beforeLink();
 beforeLink &operator =(const beforeLink &orig);
// Begin Non-Interactive User Class Declaration
public:
#include "before.non"

// End Non-Interactive User Class Declaration
};

#endif // BEFORE_HPP
```

```
// before.pre
```

```
// before.non

beforeLink(const kRepLink&, const EosRect&);
virtual void setupLink(kRepSubEpisode*, kRepSubEpisode*, tools, kRepEpisodeList*);
```

```
// contain.cpp
// User file generated by Utah.
// All user changes should be done in this file.
// This file is generated only once, it won't be regenerated.
// The full skeleton is available in contain.csk.

#ifndef CONTAIN_HPP
#include <contain.hpp>
#endif

#ifndef PROBECOM_HPP
#include <probecom.hpp>
#endif

containsLink::containsLink() : kRepLink()
{
  name="Contains";
}

containsLink::~containsLink()
{
}

containsLink &containsLink::operator =(const containsLink &orig)
```

```
{
 EOS_PROBE_LOCK(*this);
 if (this != &orig)
 {
  kRepLink::operator=(orig);
 }
 return *this;
}

containsLink::containsLink(const containsLink &orig) : kRepLink(orig)
{
}

containsLink::containsLink(const kRepLink &orig, const EosRect &frame) : kRepLink(orig)
{
  name="Contains";
  shape->setFrame(frame);
}

void containsLink::setupLink(kRepSubEpisode *from, kRepSubEpisode *to, tools id, kRepEpisodeList
*lst)
{
  kRepLink::setupLink(from, to, id, lst);
}
```

```
// contain.hpp
// Header Generated by Utah, don't tread on me.
// Portions Copyright 1994 ViewSoft, Inc. All rights reserved.
#ifdef EOS_STRICT_INCLUDE
 #ifdef CONTAIN_HPP
  #error file already included
 #endif
#endif

#ifndef CONTAIN_HPP
#define CONTAIN_HPP
// User code for this class is in contain
#ifndef EOSDEFS_HPP
#include <eosdefs.hpp>
#endif

#ifndef POINTER_HPP
#include <pointer.hpp>
#endif


///// User Pre-Declaration Code /////
#include "contain.pre"

///// End User Pre-Declaration Code /////

#ifndef containsLink_POINTER_DEFINED
```

```
#define containsLink_POINTER_DEFINED
EOS_PTR_CLASS(containsLink)
#endif

#ifndef KRLINK_HPP
#include <krlink.hpp>
#endif

class EOS_MODIFIER containsLink: public kRepLink {
public:

public:
  EOS_GET_SDO(containsLink)
  containsLink();
  containsLink(const containsLink &orig);
  virtual ~containsLink();
  containsLink &operator =(const containsLink &orig);
// Begin Non-Interactive User Class Declaration
public:
#include "contain.non"

// End Non-Interactive User Class Declaration
};

#endif // CONTAIN_HPP
```

```
// contain.pre
```

```
// contain.non

containsLink(const kRepLink&, const EosRect&);
virtual void setupLink(kRepSubEpisode*, kRepSubEpisode*, tools, kRepEpisodeList*);
```

```
// isa.cpp
// User file generated by Utah.
// All user changes should be done in this file.
// This file is generated only once, it won't be regenerated.
// The full skeleton is available in isa.csk.

#ifndef ISA_HPP
#include <isa.hpp>
#endif

#ifndef PROBECOM_HPP
#include <probecom.hpp>
#endif

isALink::isALink() : kRepLink()
```

```
{
  name="Is-A";
}

isALink::~isALink()
{
}

isALink &isALink::operator =(const isALink &orig)
{
 EOS_PROBE_LOCK(*this);
 if (this != &orig)
 {
  kRepLink::operator=(orig);
 }
 return *this;
}

isALink::isALink(const isALink &orig) : kRepLink(orig)
{
}

isALink::isALink(const kRepLink &orig, const EosRect &frame) : kRepLink(orig)
{
  name="Is-A";
  shape->setFrame(frame);
}

void isALink::setupLink(kRepSubEpisode *from, kRepSubEpisode *to, tools id, kRepEpisodeList *lst)
{
  kRepLink::setupLink(from, to, id, lst);
}
```

```
// isa.hpp
// Header Generated by Utah, don't tread on me.
// Portions Copyright 1994 ViewSoft, Inc. All rights reserved.
#ifdef EOS_STRICT_INCLUDE
 #ifdef ISA_HPP
  #error file already included
 #endif
#endif

#ifndef ISA_HPP
#define ISA_HPP
// User code for this class is in isa
#ifndef EOSDEFS_HPP
#include <eosdefs.hpp>
#endif

#ifndef POINTER_HPP
#include <pointer.hpp>
#endif
```

```
///// User Pre-Declaration Code /////
#include "isa.pre"

///// End User Pre-Declaration Code /////

#ifndef isALink_POINTER_DEFINED
#define isALink_POINTER_DEFINED
EOS_PTR_CLASS(isALink)
#endif

#ifndef KRLINK_HPP
#include <krlink.hpp>
#endif

class EOS_MODIFIER isALink: public kRepLink {
public:

public:
  EOS_GET_SDO(isALink)
  isALink();
  isALink(const isALink &orig);
  virtual ~isALink();
  isALink &operator =(const isALink &orig);
// Begin Non-Interactive User Class Declaration
public:
#include "isa.non"

// End Non-Interactive User Class Declaration
};

#endif // ISA_HPP
```

```
// isa.pre
```

```
// isa.non

isALink(const kRepLink&, const EosRect&);
virtual void setupLink(kRepSubEpisode*, kRepSubEpisode*, tools, kRepEpisodeList*);
```

```
// krdoc.cpp
// User file generated by Utah.
// All user changes should be done in this file.
// This file is generated only once, it won't be regenerated.
// The full skeleton is available in krdoc.csk.
//
// Author:Hung-Chou Tai
```

```
// Place:MIT
// Last Modified:4/2/95
// Description:

#ifndef KRDOC_HPP
#include <krdoc.hpp>
#endif

#ifndef PROBECOM_HPP
#include <probecom.hpp>
#endif

#include <acmdsele.hpp>
#include <arrobjec.hpp>
#include <probecom.hpp>
#include "kreplist.hpp"
#include "krepis.hpp"
#include "krlinkpl.hpp"
#include "krmsg.hpp"

kRepDocument::kRepDocument() : EosDocument(),
 curEpisode(new kRepEpisode()),
 newEpisode(new kRepEpisode()),
 episodeList(new kRepEpisodeList()),
 selector(eosTrue),
 linkID(new kRepLinkPalette()),
 episode(eosFalse),
 messageList(new kRepEpisodeList())
{
  episodeList->sSelect.addProbe(EOS_PROBE(kRepDocument, episodeSelected, EosSignal&), this);
  episodeList->sRemove.addProbe(EOS_PROBE(kRepDocument, episodeRemoved, EosSignal&), this);
}

kRepDocument::~kRepDocument()
{
 delete (kRepEpisode *)curEpisode;
 delete (kRepEpisode *)newEpisode;
 delete (kRepEpisodeList *)episodeList;
 delete (kRepLinkPalette *)linkID;
 delete (kRepEpisodeList *)messageList;
}

kRepDocument &kRepDocument::operator =(const kRepDocument &orig)
{
 EOS_PROBE_LOCK(*this);
 if (this != &orig)
 {
  EosDocument::operator=(orig);
  delete (kRepEpisode *)curEpisode;
  curEpisode = new kRepEpisode(*orig.curEpisode);
  delete (kRepEpisode *)newEpisode;
  newEpisode = new kRepEpisode(*orig.newEpisode);
  delete (kRepEpisodeList *)episodeList;
  episodeList = new kRepEpisodeList(*orig.episodeList);
```

```
    selector = orig.selector;
    delete (kRepLinkPalette *)linkID;
    linkID = new kRepLinkPalette(*orig.linkID);
    episode = orig.episode;
    delete (kRepEpisodeList *)messageList;
    messageList = new kRepEpisodeList(*orig.messageList);
   }
  return *this;
}

kRepDocument::kRepDocument(const kRepDocument &orig) : EosDocument(orig),
  curEpisode(new kRepEpisode(*orig.curEpisode)),
  newEpisode(new kRepEpisode(*orig.newEpisode)),
  episodeList(new kRepEpisodeList(*orig.episodeList)),
  selector(orig.selector),
  linkID(new kRepLinkPalette(*orig.linkID)),
  episode(orig.episode),
  messageList(new kRepEpisodeList(*orig.messageList))
{
}

void kRepDocument::addEpisode(kRepEpisode *arg)
{
}

EosBoolean kRepDocument::addEpisode_init(kRepEpisode *arg)
{
  arg=new kRepEpisode();
  selector=eosFalse;
  episode=eosTrue;
  return eosTrue;
}

EosBoolean kRepDocument::addEpisode_validate(kRepEpisode *arg,EosShell *shell)
{
  return eosTrue;
}

void kRepDocument::episodeSelected(EosSignal &signal)
{
  EosArraySelectCommand* selCommand =
          (EosArraySelectCommand *)episodeList->getArrayCommand();
  EosIndex curIndex=selCommand->getIndex();
  curEpisode=(kRepEpisode*) episodeList->getObject(curIndex);
}

void kRepDocument::resetTools()
{
   selector=eosTrue;
   episode=eosFalse;
}

void kRepDocument::episodeRemoved(EosSignal &signal)
{
```

```
  curEpisode=new kRepEpisode();
}

void kRepDocument::receiveMessage(kRepMessage *msg)
{
  messageList->insertObject(msg, 0);
}
```

```
// krdoc.hpp
// Header Generated by Utah, don't tread on me.
// Portions Copyright 1994 ViewSoft, Inc. All rights reserved.
#ifdef EOS_STRICT_INCLUDE
 #ifdef KRDOC_HPP
  #error file already included
 #endif
#endif

#ifndef KRDOC_HPP
#define KRDOC_HPP
// User code for this class is in krdoc
#ifndef EOSDEFS_HPP
#include <eosdefs.hpp>
#endif

#ifndef POINTER_HPP
#include <pointer.hpp>
#endif


///// User Pre-Declaration Code /////
#include "krdoc.pre"

///// End User Pre-Declaration Code /////

#ifndef kRepDocument_POINTER_DEFINED
#define kRepDocument_POINTER_DEFINED
EOS_PTR_CLASS(kRepDocument)
#endif

#ifndef DOCUMENT_HPP
#include <document.hpp>
#endif

#ifndef kRepEpisode_POINTER_DEFINED
#define kRepEpisode_POINTER_DEFINED
EOS_PTR_CLASS(kRepEpisode)
#endif

#ifndef kRepEpisodeList_POINTER_DEFINED
#define kRepEpisodeList_POINTER_DEFINED
EOS_PTR_CLASS(kRepEpisodeList)
#endif
```

```
#ifndef kRepLinkPalette_POINTER_DEFINED
#define kRepLinkPalette_POINTER_DEFINED
EOS_PTR_CLASS(kRepLinkPalette)
#endif

#ifndef EOSBOOL_HPP
#include <eosbool.hpp>
#endif

class EOS_MODIFIER kRepDocument: public EosDocument {
public:
  virtual EosBoolean addEpisode_init(kRepEpisode *arg);
  virtual EosBoolean addEpisode_validate(kRepEpisode *arg, EosShell *shell);

  virtual void addEpisode(kRepEpisode *arg);
public:
  kRepEpisodePtr curEpisode;
  kRepEpisodePtr newEpisode;
  kRepEpisodeListPtr episodeList;
  EosBool selector;
  kRepLinkPalettePtr linkID;
  EosBool episode;
  kRepEpisodeListPtr messageList;
  EOS_GET_SDO(kRepDocument)
  kRepDocument();
  kRepDocument(const kRepDocument &orig);
  virtual ~kRepDocument();
  kRepDocument &operator =(const kRepDocument &orig);
// Begin Non-Interactive User Class Declaration
public:
#include "krdoc.non"

// End Non-Interactive User Class Declaration
};

#endif // KRDOC_HPP
```

```
// krdoc.pre

class kRepMessage;

#include "krlinkpl.hpp"

enum tools
{
   null=0,
   lifeline=1,
   sameAs=2,
   notSameAs=3,
   contains=4,
   notContains=5,
```

```
  before=6,
  after=7,
  isA=8,
  notIsA=9,
  notAfter=10,
  notBefore=11
};

enum direction
{
  none=0,
  origination=1,
  destination=2
};
```

```
// krdoc.non

kRepEpisode* getNewEpisode() { return newEpisode; }
kRepEpisode* getCurEpisode() { return curEpisode; }
tools getToolId() { return (tools) (int) linkID->linkID; }
void resetTools();
EosBool selectorSelected() { return selector; }
EosBool participantSelected() { return episode; }
kRepEpisodeList *getEpisodeList() { return episodeList; }

void episodeSelected(EosSignal &);
void episodeRemoved(EosSignal &);

virtual void receiveMessage(kRepMessage *);
```

```
// krdrawep.cpp
//
// Author:Hung-Chou Tai
// Place:MIT
// Last Modified:4/2/95
// Description:
//    Command object for drawing episodes on the screen.
//    See krdrawep.hpp for member function declarations.

#include "krdrawep.hpp"

#include "krepis.hpp"
#include "kreplist.hpp"
#include "krview.hpp"
#include "krframe.hpp"
#include "krshape.hpp"
#include "krdoc.hpp"

extern EosFramework *mainFrame;
```

```
kRepDrawEpisode::kRepDrawEpisode(kRepEpisode *argEpis,
  kRepView *argView, const EosPoint &theMouse, EosSystemEvent::
  ButtonSelector buttonSelector, EosBoolean constrainsMouse,
  EosBoolean constrainToView,
  EosBoolean trackNonMovement) :
  EosMouseCommand(drawCommand, argView, theMouse, buttonSelector,
    constrainsMouse, constrainToView, trackNonMovement),
  curEpisode(argEpis),
  curShape(argEpis->getShape()),
  curFrame(theMouse, theMouse),
  curView(argView),
  insertPosition(-1)
{
}

void kRepDrawEpisode::trackFeedback(TrackPhase aTrackPhase,
  const EosPoint& anchorPoint, const EosPoint& previousPoint,
  EosPoint& nextPoint, EosBoolean mouseMoved, EosBoolean turnItOn)
{
  EosColor black(0,0,0);
  curView->setLineColor(black);
  curView->setLineThickness(1);
  curView->setLineHeight(0);
  curView->setFilledHeight(0);
  curView->setDisplayFilled(eosFalse);
  curView->setLineMode(eosNotXorPen);
  curView->setLineType(eosDotLine);
  curEpisode->drawFeedback(*curView, curFrame);
}

EosMouseCommand* kRepDrawEpisode::trackMouse(TrackPhase aTrackPhase,
  const EosPoint& anchorPoint, const EosPoint& previousPoint,
  EosPoint &nextPoint, EosBoolean mouseMoved)
{
  if(mouseMoved)
  {
    curFrame = EosRect(anchorPoint, nextPoint);
    curFrame.validate();
    curShape->setFrame(curFrame);
  }
  return this;
}

void kRepDrawEpisode::doIt()
{
  kRepEpisodeList* episodeList=curView->getEpisodeList();
  kRepFrame* frame=(kRepFrame*)mainFrame;
  kRepDocument* document=(kRepDocument*) frame->getCurrentDocument();
  kRepEpisode *parent=curEpisode->findContainerEpisode(episodeList);

  EosMouseCommand::doIt();
  if(parent)
    parent->addEpisode(curEpisode);
```

```
  curEpisode->addContainedEpisodes(episodeList);
  insertPosition=episodeList->addObject(curEpisode);
  episodeList->setData(curEpisode, insertPosition);
  episodeList->select(eosTrue, insertPosition);
  document->resetTools();
}

void kRepDrawEpisode::undoIt()
{
  EosMouseCommand::undoIt();
  kRepEpisodeList* episodeList=curView->getEpisodeList();
  episodeList->setData(curEpisode, eosTrue);
  if(insertPosition >=0)
    episodeList->remove(insertPosition);
}

void kRepDrawEpisode::commit()
{
}
```

```
// krdrawep.hpp

#ifndef KRDRAWEP_HPP
#define KRDRAWEP_HPP

#include <mousecmd.hpp>
#include <rect.hpp>
#include <sysevent.hpp>
#include <viewcmd.hpp>
#include <eosint.hpp>

class kRepShape;
class kRepView;
class kRepEpisode;

class EOS_MODIFIER kRepDrawEpisode : public EosMouseCommand
{
  public:
    enum { drawCommand=5001 };
    kRepDrawEpisode(kRepEpisode *, kRepView *, const EosPoint&,
      EosSystemEvent::ButtonSelector buttonSelector=EosSystemEvent::kDefaultButton,
      EosBoolean constrainsMouse=eosFalse,
      EosBoolean constrainToView=eosFalse,
      EosBoolean trackNonMovement=eosFalse);
    ~kRepDrawEpisode() {}
    virtual void trackFeedback(TrackPhase, const EosPoint&,
      const EosPoint&, EosPoint&, EosBoolean, EosBoolean);
    virtual EosMouseCommand* trackMouse(TrackPhase,
      const EosPoint&, const EosPoint&, EosPoint &,
      EosBoolean);
    virtual void doIt();
    virtual void undoIt();
```

```
    virtual void redoIt() { doIt(); }
    virtual void commit();

  protected:
    kRepView *curView;
    kRepEpisode *curEpisode;
    kRepShape *curShape;
    EosRect curFrame;
    EosInt insertPosition;
};
#endif
```

```
// krdrawlk.cpp
//
// Author:Hung-Chou Tai
// Place:MIT
// Last Modified:4/2/95
// Description:
//    Command object for drawing links on the screen.
//    See krdrawep.hpp for member function declarations.

#include "krdrawlk.hpp"

#include "krlink.hpp"
#include "krview.hpp"
#include "krepis.hpp"
#include "kreplist.hpp"
#include "krdoc.hpp"
#include "krsbepis.hpp"
#include "links.hpp"

kRepDrawLink::kRepDrawLink(kRepLink *newLink, kRepEpisode *click, kRepView *argView, tools
link,
  const EosPoint &theMouse, EosSystemEvent::ButtonSelector buttonSelector,
  EosBoolean constrainsMouse, EosBoolean constrainToView,
  EosBoolean trackNonMovement) :
  EosMouseCommand(drawCommand, argView, theMouse, buttonSelector,
    constrainsMouse, constrainToView, trackNonMovement),
  curView(argView),
  fromEpisode(click),
  toEpisode(NULL),
  episodeList(argView->getEpisodeList()),
  curFrame(theMouse, theMouse),
  insertPosition(-1),
  linkType(link)
{
  initCurLink(newLink, theMouse, linkType);
  curShape=curLink->getShape();
}

void kRepDrawLink::trackFeedback(TrackPhase aTrackPhase,
  const EosPoint& anchorPoint, const EosPoint& previousPoint,
```

```
  EosPoint& nextPoint, EosBoolean mouseMoved, EosBoolean turnItOn)
{
  EosColor black(0,0,0);
  curView->setLineColor(black);
  curView->setLineThickness(1);
  curView->setLineHeight(0);
  curView->setFilledHeight(0);
  curView->setDisplayFilled(eosFalse);
  curView->setLineMode(eosNotXorPen);
// curView->setLineMode(eosXor);
  curView->setLineType(eosDotLine);
  curLink->drawFeedback(*curView, curFrame);
}

EosMouseCommand* kRepDrawLink::trackMouse(TrackPhase aTrackPhase,
  const EosPoint& anchorPoint, const EosPoint& previousPoint,
  EosPoint &nextPoint, EosBoolean mouseMoved)
{
  if(mouseMoved)
  {
    curFrame.validate();
    curShape->setBegin(anchorPoint);
    curShape->setEnd(nextPoint);
  }
  return this;
}

void kRepDrawLink::doIt()
{
  kRepSubEpisode *from, *to;
  EosIndex i;
  EosMouseCommand::doIt();
  toEpisode=curView->pointInEpisode(fLastNextPoint);

  if((fromEpisode==toEpisode)||(toEpisode==NULL))
  {
    curShape->invalidate(*curView);
    return;
  }
  EosPoint begin=fAnchorPoint;
  EosPoint end=fLastNextPoint;
// Flush the begin and end points of the link with the sides
// of the rectangles
  curLink->makePretty(begin, end, fromEpisode, toEpisode);
  from=new kRepSubEpisode(EosRect(begin, begin).inflate(EosPoint(SUBEPSIZE, SUBEPSIZE)), link-
                                        Type, eosFalse, origination, "->"+curLink->getName(), NULL);
  to=new kRepSubEpisode(EosRect(end, end).inflate(EosPoint(SUBEPSIZE, SUBEPSIZE)), linkType,
                                    eosFalse, destination, curLink->getName()+"->", NULL);
  curLink->setupLink(from, to, linkType, episodeList);
  // First, update the episodes being linked.
  fromEpisode->addEpisode(from);
  toEpisode->addEpisode(to);
  // Then, update the entire view area.
  episodeList->addObject(from);
```

```
  episodeList->addObject(to);
  insertPosition=episodeList->addObject(curLink);
  episodeList->setData(curLink, insertPosition);
  episodeList->select(eosTrue, insertPosition);

  if(!fromEpisode->allowSubEpisode(from, episodeList))
  {
    i=episodeList->findEpisode(curLink);
    episodeList->remove(i);
    curLink->removeAssociations(episodeList, eosTrue);
    curView->invalidateView(EosRect(EosPoint(0,0),
      EosPoint(curView->maxDrawableWidth(), curView->maxDrawableHeight())), eosTrue);
    delete curLink;
  }
  else
    if(!toEpisode->allowSubEpisode(to, episodeList))
    {
      i=episodeList->findEpisode(curLink);
      episodeList->remove(i);
      curLink->removeAssociations(episodeList, eosTrue);
      curView->invalidateView(EosRect(EosPoint(0,0),
        EosPoint(curView->maxDrawableWidth(), curView->maxDrawableHeight())), eosTrue);
      delete curLink;
    }
    else
    {
      from->deriveNew(episodeList);
      to->deriveNew(episodeList);
    }

  curView->invalidateView(EosRect(EosPoint(0,0),
    EosPoint(curView->maxDrawableWidth(), curView->maxDrawableHeight())), eosTrue);
}

void kRepDrawLink::undoIt()
{
  EosMouseCommand::undoIt();
  episodeList->setData(curLink, eosTrue);

  if(insertPosition >=0)
    episodeList->remove(insertPosition);
}

void kRepDrawLink::commit()
{
}

void kRepDrawLink::initCurLink(kRepLink *newLink, EosPoint pt, tools id)
{
  EosRect frame(pt, pt);
  switch(id)
  {
    case lifeline:
      curLink=new kRepLifeline(*newLink, frame);
```

```
      break;
    case sameAs:
      curLink=new sameAsLink(*newLink, frame);
      break;
    case notSameAs:
      curLink=new notSameAsLink(*newLink, frame);
      break;
    case contains:
      curLink=new containsLink(*newLink, frame);
      break;
    case notContains:
      curLink=new notContainsLink(*newLink, frame);
      break;
    case before:
      curLink=new beforeLink(*newLink, frame);
      break;
    case after:
      curLink=new afterLink(*newLink, frame);
      break;
    case isA:
      curLink=new isALink(*newLink, frame);
      break;
    case notIsA:
      curLink=new notIsALink(*newLink, frame);
      break;
    case notBefore:
      curLink=new notBeforeLink(*newLink, frame);
      break;
    case notAfter:
      curLink=new notAfterLink(*newLink, frame);
      break;
    case null:
    default:
      curLink=newLink;
      break;
  }
}
```

```
// krdrawlk.hpp

#ifndef KRDRAWLK_HPP
#define KRDRAWLK_HPP

#include <mousecmd.hpp>
#include <rect.hpp>
#include <sysevent.hpp>
#include <viewcmd.hpp>
#include <eosint.hpp>
#include "krdoc.hpp"

class kRepLink;
class kRepShape;
```

```
class kRepView;
class kRepEpisode;
class kRepEpisodeList;

class EOS_MODIFIER kRepDrawLink : public EosMouseCommand
{
  public:
    enum { drawCommand=5001 };
    kRepDrawLink(kRepLink *, kRepEpisode *, kRepView *, tools, const EosPoint&,
      EosSystemEvent::ButtonSelector buttonSelector=EosSystemEvent::kDefaultButton,
      EosBoolean constrainsMouse=eosFalse,
      EosBoolean constrainToView=eosFalse,
      EosBoolean trackNonMovement=eosFalse);
    ~kRepDrawLink() { }
    virtual void trackFeedback(TrackPhase, const EosPoint&,
      const EosPoint&, EosPoint&, EosBoolean, EosBoolean);
    virtual EosMouseCommand* trackMouse(TrackPhase,
      const EosPoint&, const EosPoint&, EosPoint &,
      EosBoolean);
    virtual void doIt();
    virtual void undoIt();
    virtual void redoIt() { doIt(); }
    virtual void commit();
    virtual void initCurLink(kRepLink*, EosPoint, tools);
  protected:
    kRepView *curView;
    kRepLink *curLink;
    kRepEpisode *fromEpisode;
    kRepEpisode *toEpisode;
    kRepEpisodeList *episodeList;
    kRepShape *curShape;
    EosRect curFrame;
    EosInt insertPosition;
    tools linkType;
};
#endif
```

```
// krepis.cpp
// User file generated by Utah.
// All user changes should be done in this file.
// This file is generated only once, it won't be regenerated.
// The full skeleton is available in krepis.csk.
//
// Author:Hung-Chou Tai
// Place:MIT
// Last Modified:4/2/95
// Description:
//    Episode class member function definitions.
//    Check krepis.hpp, krepis.pre, krepis.non
//    for declarations.

#ifndef KREPIS_HPP
```

```
#include <krepis.hpp>
#endif

#ifndef PROBECOM_HPP
#include <probecom.hpp>
#endif

#include <editarg.hpp>
#include <message.hpp>
#include "krshape.hpp"
#include "krrect.hpp"
#include "kreplist.hpp"
#include "krsbepis.hpp"

#define OFFSET 14
#define EPISODEEDITVIEW "Edit episode view"

kRepEpisode::kRepEpisode() : EosObject(),
 name(),
 shape(new kRepRectangle()),
 episodeList(new kRepEpisodeList()),
 selected(eosFalse),
 instance(),
 showInstance(eosFalse),
 parentDeletable(eosFalse),
 selfControl(eosTrue),
 creatorEpisode(NULL)
{
}

kRepEpisode::~kRepEpisode()
{
 delete (kRepShape *)shape;
// delete (kRepEpisodeList *)episodeList;
}

kRepEpisode &kRepEpisode::operator =(const kRepEpisode &orig)
{
 EOS_PROBE_LOCK(*this);
 if (this != &orig)
 {
  EosObject::operator=(orig);
  name = orig.name;
  delete (kRepShape *)shape;
  shape = new kRepRectangle(*orig.shape);
  delete (kRepEpisodeList *)episodeList;
  episodeList = new kRepEpisodeList(*orig.episodeList);
  selected = orig.selected;
  instance = orig.instance;
  showInstance = orig.showInstance;
  parentDeletable=orig.parentDeletable;
  selfControl=orig.selfControl;
  creatorEpisode=orig.creatorEpisode;
 }
```

```
 return *this;
}

kRepEpisode::kRepEpisode(const kRepEpisode &orig) : EosObject(orig),
 name(orig.name),
 shape(new kRepRectangle(*orig.shape)),
 episodeList(new kRepEpisodeList(*orig.episodeList)),
 selected(orig.selected),
 instance(orig.instance),
 showInstance(orig.showInstance),
 parentDeletable(orig.parentDeletable),
 selfControl(orig.selfControl),
 creatorEpisode(orig.creatorEpisode)
{
}

kRepEpisode::kRepEpisode(const kRepEpisode &orig, const EosRect &frame) :
   name(orig.name),
   shape(new kRepRectangle(frame)),
   episodeList(new kRepEpisodeList(*orig.episodeList)),
   selected(orig.selected),
   instance(orig.instance),
   showInstance(orig.showInstance),
   parentDeletable(orig.parentDeletable),
   selfControl(orig.selfControl),
   creatorEpisode(orig.creatorEpisode)
{
}

void kRepEpisode::draw(kRepView &view)
{
  EosColor black(0,0,0);
  EosPoint upperLeft=getUpperLeft()-EosPoint(0, OFFSET);
  view.setTextColor(black);
  view.setTextHeight(0);
  view.setDisplayFilled(eosFalse);
  shape->draw(view);
  if(showInstance)
    view.drawText(name+": "+instance, upperLeft, -1);
  else
    view.drawText(name, upperLeft, -1);
}

void kRepEpisode::drawFeedback(kRepView &view, const EosRect &frame)
{
  EosColor black(0,0,0);
  EosPoint upperLeft=getUpperLeft()-EosPoint(0, OFFSET);
  view.setTextColor(black);
  view.setTextHeight(1);
  shape->drawFeedback(view, frame);
  if(showInstance)
    view.drawText(name+": "+instance, upperLeft, -1);
  else
    view.drawText(name, upperLeft, -1);
```

```
}

void kRepEpisode::invalidate(kRepView &view)
{
  shape->invalidate(view);
}

EosRect &kRepEpisode::getFrame()
{
  return shape->getFrame();
}

EosPoint kRepEpisode::getUpperLeft()
{
  return shape->getUpperLeft();
}

void kRepEpisode::setSelect(EosBoolean sel, kRepView *view)
{
  selected=sel;
  if(view)
    setFilled(sel, *view);
}

void kRepEpisode::setFilled(EosBoolean bool, kRepView &view)
{
// view.setDisplayFilled(bool);
}

EosBoolean kRepEpisode::containsPoint(const EosPoint& point)
{
  return shape->containsPoint(point);
}

// This function needs to remove all pointer references
// including all episodes referenced from this one from
// the list given.
void kRepEpisode::removeAssociations(kRepEpisodeList *lst, EosBoolean del)
{
  EosIndex i,j;

  kRepEpisode *parent=findParent(lst);
  if(parent&&del)
    parent->removeEpisode(this);

  for (i=episodeList->getLength()-1; i>=0; i--)
  {
    kRepEpisode *thisEpisode=(kRepEpisode*) episodeList->getObject(i);
    if(!thisEpisode->selfControllable())
    {
      j=lst->findEpisode(thisEpisode);
      if(j>=0)
      {
        if(del)
```

```
            episodeList->remove(i);
          lst->remove(j);
          thisEpisode->removeAssociations(lst, del);
          if(del)
            delete thisEpisode;
        }
      }
    }
}

EosBoolean kRepEpisode::removeSurrounding(kRepEpisodeList *lst)
{
  EosRect frame;
  shape->getFrame(frame);

  EosIndex i;
  EosMessage message(0L, eosQuestion, eosYesNo, eosApplication,
    "Double clicking will erase all but the clicked object. Continue?",
    "Double Click", 1);

  if(message.display()==eosMessageNo)
    return eosFalse;
  for(i=lst->getLength()-1; i>=0; i--)
  {
    kRepEpisode *cur=(kRepEpisode *) lst->getObject(i);
    if(cur!=this)
      if(!frame.contains(cur->getFrame()))
      {
        cur->removeAssociations(lst, eosTrue);
        lst->remove(i);
        delete cur;
      }
  }
  return eosTrue;
}

EosBoolean kRepEpisode::isParentOf(kRepEpisode *ep)
{
  kRepEpisodeListIter iter(*episodeList);
  while(!iter.isDone())
  {
    if(iter.getCurrent()==ep)
      return eosTrue;
    iter.next();
  }
  return eosFalse;
}

EosBoolean kRepEpisode::containsEpisode(kRepEpisode *ep)
{
  EosRect frame=getFrame();
  return frame.contains(ep->getFrame());
}
```

```
void kRepEpisode::addContainedEpisodes(kRepEpisodeList *list)
{
  kRepEpisodeListIter iter(*list);
  kRepEpisode *cur, *curParent;
  kRepShape *curShape;
  EosRect curFrame, thisFrame;
  EosIndex i;

  shape->getFrame(thisFrame);

  while(!iter.isDone())
  {
    cur=iter.getCurrent();
    if(cur->selfControllable())
    {
      if(containsEpisode(cur))
      {
        curParent=cur->findContainerEpisode(episodeList);
        for(i=episodeList->getLength()-1; i>=0; i--)
        {
          kRepEpisode *tmpCur=
            (kRepEpisode*) episodeList->getObject(i);
          if(cur->containsEpisode(tmpCur))
            removeEpisode(tmpCur);
        }
        if(!curParent)
          addEpisode(cur);
      }
    }
    iter.next();
  }
}

void kRepEpisode::addEpisode(kRepEpisode *arg)
{
  episodeList->addObject(arg);
}

void kRepEpisode::addEpisode(kRepEpisode *arg, int i)
{
  episodeList->insertObject(arg, i);
}

kRepEpisode* kRepEpisode::findParent(kRepEpisodeList *lst)
{
  kRepEpisodeListIter iter(*lst);
  kRepEpisode *cur;
  while (!iter.isDone())
  {
    cur = (kRepEpisode*) iter.getCurrent();
    if(cur->isParentOf(this))
      return cur;
    iter.next();
  }
```

```
  return NULL;
}

kRepEpisode *kRepEpisode::findContainerEpisode(kRepEpisodeList *lst)
{
  kRepEpisodeListIter iter(*lst);
  kRepEpisode *ret=NULL, *cur;
  kRepShape *curShape;
  EosRect frame, thisFrame;

  shape->getFrame(thisFrame);

  while (!iter.isDone())
  {
    cur = (kRepEpisode*) iter.getCurrent();
    curShape = cur->getShape();
    curShape->getFrame(frame);
    if(frame.contains(thisFrame)&&(ret==NULL))
      ret=cur;
    else
      if((ret!=NULL)&&frame.contains(thisFrame)&&ret->containsEpisode(cur))
        ret=cur;
    iter.next();
  }
  return ret;
}

void kRepEpisode::parentIsDeletable(EosBoolean bool)
{
  parentDeletable=bool;
}

EosBoolean kRepEpisode::allowSubEpisode(kRepSubEpisode *test, kRepEpisodeList *lst)
{
  kRepEpisodeListIter iter(*episodeList);
  kRepEpisode *cur;
  EosBoolean ret=eosTrue;
  while (!iter.isDone())
  {
    cur=iter.getCurrent();
    if(!cur->selfControllable())
      if(!cur->allowSubEpisode(test, lst))
        return eosFalse;
    iter.next();
  }
  return eosTrue;
}

EosBoolean kRepEpisode::removeEpisode(kRepEpisode *arg)
{
  EosIndex i=episodeList->findEpisode(arg);
  if(i>=0)
  {
    episodeList->remove(i);
```

```
    return eosTrue;
  }
  else
    return eosFalse;
}

int kRepEpisode::findID(tools id, direction d, kRepEpisodeList *lst)
{
  int i=0;
  kRepEpisodeListIter iter(*episodeList);
  while (!iter.isDone())
  {
    kRepSubEpisode *cur=(kRepSubEpisode *) iter.getCurrent();
    if(!cur->selfControllable())
      if(cur->getLinkID()==id||id==null)
        if(cur->getDirection()==d||d==none)
        {
          i++;
          lst->addObject(cur);
        }
    iter.next();
  }
  return i;
}

void kRepEpisode::edit(kRepView &view)
{
  EosAtom *editView=new EosAtom(EPISODEEDITVIEW);
  EosEditArg *editArg=new EosEditArg(*editView);
  editWait(*editArg);
}

void kRepEpisode::selfControllable(EosBoolean b)
{
  selfControl=b;
}
```

```
// krepis.hpp
// Header Generated by Utah, don't tread on me.
// Portions Copyright 1994 ViewSoft, Inc. All rights reserved.
#ifdef EOS_STRICT_INCLUDE
 #ifdef KREPIS_HPP
  #error file already included
 #endif
#endif

#ifndef KREPIS_HPP
#define KREPIS_HPP
// User code for this class is in krepis
#ifndef EOSDEFS_HPP
#include <eosdefs.hpp>
#endif
```

```
#ifndef POINTER_HPP
#include <pointer.hpp>
#endif


///// User Pre-Declaration Code /////
#include "krepis.pre"

///// End User Pre-Declaration Code /////

#ifndef kRepEpisode_POINTER_DEFINED
#define kRepEpisode_POINTER_DEFINED
EOS_PTR_CLASS(kRepEpisode)
#endif

#ifndef OBJECT_HPP
#include <object.hpp>
#endif

#ifndef kRepShape_POINTER_DEFINED
#define kRepShape_POINTER_DEFINED
EOS_PTR_CLASS(kRepShape)
#endif

#ifndef kRepEpisodeList_POINTER_DEFINED
#define kRepEpisodeList_POINTER_DEFINED
EOS_PTR_CLASS(kRepEpisodeList)
#endif

#ifndef STRING_HPP
#include <string.hpp>
#endif

#ifndef EOSBOOL_HPP
#include <eosbool.hpp>
#endif

class EOS_MODIFIER kRepEpisode: public EosObject {
public:

public:
  EosString name;
  kRepShapePtr shape;
  kRepEpisodeListPtr episodeList;
  EosBool selected;
  EosString instance;
  EosBool showInstance;
  EOS_GET_SDO(kRepEpisode)
  kRepEpisode();
  kRepEpisode(const kRepEpisode &orig);
  virtual ~kRepEpisode();
  kRepEpisode &operator =(const kRepEpisode &orig);
// Begin Non-Interactive User Class Declaration
```

```
public:
#include "krepis.non"

// End Non-Interactive User Class Declaration
};

#endif // KREPIS_HPP
```

```
// krepis.pre

class kRepSubEpisode;

//enum tools;
//enum direction;
#include "krview.hpp"
#include "krdoc.hpp"
#include "krshape.hpp"
#include "kreplist.hpp"
#include "rect.hpp"
```

```
// krepis.non

// Episode flags.
EosBoolean parentDeletable;
EosBoolean selfControl;
kRepEpisode *creatorEpisode;

kRepEpisode(const kRepEpisode &, const EosRect &);

// Viewing Functions
virtual void draw(kRepView &);
virtual void drawFeedback(kRepView &, const EosRect &);
virtual void invalidate(kRepView &);
virtual void setFilled(EosBoolean, kRepView &);
virtual void setSelect(EosBoolean, kRepView *);

// Shape Functions
kRepEpisode *getCreator() { return creatorEpisode; }
virtual kRepShape* getShape(void) { return shape; }
EosRect &getFrame(void);
EosPoint getUpperLeft();

// Screen testing functions to check for physical boundaries
virtual EosBoolean containsPoint(const EosPoint&);
virtual EosBoolean containsEpisode(kRepEpisode*);
virtual void addContainedEpisodes(kRepEpisodeList*);
kRepEpisode* findContainerEpisode(kRepEpisodeList *);

// Episode functions
virtual EosBoolean isParentOf(kRepEpisode*);
```

```
EosBoolean isSelected() { return selected; }
virtual void removeAssociations(kRepEpisodeList*, EosBoolean);
virtual EosBoolean removeSurrounding(kRepEpisodeList*);
//virtual EosBoolean removeAll(kRepEpisode *, kRepEpisodeList *) { return eosFalse; }
kRepEpisode* findParent(kRepEpisodeList*);
virtual void addEpisode(kRepEpisode *);
virtual void addEpisode(kRepEpisode *, int);
virtual EosBoolean removeEpisode(kRepEpisode *);
virtual int findID(tools, direction, kRepEpisodeList *);
virtual EosBoolean allowSubEpisode(kRepSubEpisode *, kRepEpisodeList *);

// Helper functions for member data acess
EosString getName() { return name; }
kRepEpisodeList* getEpisodeList() { return episodeList; }
void parentIsDeletable(EosBoolean);
void selfControllable(EosBoolean);
EosBoolean selfControllable() { return selfControl; }
virtual void edit(kRepView &);
```

```
// kreplist.cpp
// User file generated by Utah.
// All user changes should be done in this file.
// This file is generated only once, it won't be regenerated.
// The full skeleton is available in kreplist.csk.
//
// Author:Hung-Chou Tai
// Place:MIT
// Last Modified:4/2/95
// Description:
//    Episode list and list iterator member function definitions.
//    Check kreplist.hpp, kreplist.pre, and kreplist.non
//    for declarations.

#ifndef KREPLIST_HPP
#include <kreplist.hpp>
#endif

#ifndef PROBECOM_HPP
#include <probecom.hpp>
#endif

#include "krepis.hpp"

kRepEpisodeListIter::kRepEpisodeListIter(kRepEpisodeList &list) :
  episodeList(list),
  maxElements(list.getLength()),
  curElement(0)
{
// next();
}

kRepEpisode* kRepEpisodeListIter::getCurrent(void)
```

```
{
  return (kRepEpisode *) episodeList.getObject(curElement);
}


kRepEpisodeList::kRepEpisodeList() : EosObjectArray()
{
  setOwnsElements(eosFalse);
}

kRepEpisodeList::~kRepEpisodeList()
{
}

kRepEpisodeList &kRepEpisodeList::operator =(const kRepEpisodeList &orig)
{
 EOS_PROBE_LOCK(*this);
 if (this != &orig)
 {
  EosObjectArray::operator=(orig);
  setOwnsElements(eosFalse);
 }
 return *this;
}

kRepEpisodeList::kRepEpisodeList(const kRepEpisodeList &orig) : EosObjectArray(orig)
{
  setOwnsElements(eosFalse);
}

void kRepEpisodeListIter::next(void)
{
  if(curElement < maxElements)
    curElement++;
}

EosIndex kRepEpisodeList::findEpisode(const kRepEpisode *arg)
{
  kRepEpisodeListIter iter(*this);
  while(!iter.isDone())
  {
    if(iter.getCurrent() == arg)
      return iter.getCurrentIndex();
    iter.next();
  }
  return -1;
}
```

```
// kreplist.hpp
// Header Generated by Utah, don't tread on me.
// Portions Copyright 1994 ViewSoft, Inc. All rights reserved.
#ifdef EOS_STRICT_INCLUDE
```

```
 #ifdef KREPLIST_HPP
  #error file already included
 #endif
#endif

#ifndef KREPLIST_HPP
#define KREPLIST_HPP
// User code for this class is in kreplist
#ifndef EOSDEFS_HPP
#include <eosdefs.hpp>
#endif

#ifndef POINTER_HPP
#include <pointer.hpp>
#endif


///// User Pre-Declaration Code /////
#include "kreplist.pre"

///// End User Pre-Declaration Code /////

#ifndef kRepEpisodeList_POINTER_DEFINED
#define kRepEpisodeList_POINTER_DEFINED
EOS_PTR_CLASS(kRepEpisodeList)
#endif

#ifndef ARROBJEC_HPP
#include <arrobjec.hpp>
#endif

class EOS_MODIFIER kRepEpisodeList: public EosObjectArray {
public:

public:
 EOS_GET_SDO(kRepEpisodeList)
 kRepEpisodeList();
 kRepEpisodeList(const kRepEpisodeList &orig);
 virtual ~kRepEpisodeList();
 kRepEpisodeList &operator =(const kRepEpisodeList &orig);
// Begin Non-Interactive User Class Declaration
public:
#include "kreplist.non"

// End Non-Interactive User Class Declaration
};

#endif // KREPLIST_HPP
```

```
// kreplist.pre

class kRepEpisode;
```

```
class kRepEpisodeList;

class kRepEpisodeListIter
{
  public:
    kRepEpisodeListIter(kRepEpisodeList&);
    ~kRepEpisodeListIter() {}
    void next(void);
    EosBoolean isDone(void) { return curElement>=maxElements; }
    kRepEpisode* getCurrent(void);
    EosIndex getCurrentIndex() { return curElement; }

  protected:
    int maxElements;
    int curElement;
    kRepEpisodeList& episodeList;
};
```

```
// kreplist.non

EosIndex findEpisode(const kRepEpisode*);
```

```
// krframe.cpp
// User file generated by Utah.
// All user changes should be done in this file.
// This file is generated only once, it won't be regenerated.
// The full skeleton is available in krframe.csk.
//
// Author:Hung-Chou Tai
// Place:MIT
// Last Modified:4/2/95
// Description:

#ifndef KRFRAME_HPP
#include <krframe.hpp>
#endif

#ifndef PROBECOM_HPP
#include <probecom.hpp>
#endif

kRepFrame::kRepFrame() : EosMDIFramework()
{
}

kRepFrame::~kRepFrame()
{
}

kRepFrame &kRepFrame::operator =(const kRepFrame &orig)
```

```
{
 EOS_PROBE_LOCK(*this);
 if (this != &orig)
 {
  EosMDIFramework::operator=(orig);
 }
 return *this;
}

kRepFrame::kRepFrame(const kRepFrame &orig) : EosMDIFramework(orig)
{
}
```

```
// krframe.hpp
// Header Generated by Utah, don't tread on me.
// Portions Copyright 1994 ViewSoft, Inc. All rights reserved.
#ifdef EOS_STRICT_INCLUDE
 #ifdef KRFRAME_HPP
  #error file already included
 #endif
#endif

#ifndef KRFRAME_HPP
#define KRFRAME_HPP
// User code for this class is in krframe
#ifndef EOSDEFS_HPP
#include <eosdefs.hpp>
#endif

#ifndef POINTER_HPP
#include <pointer.hpp>
#endif

#ifndef kRepFrame_POINTER_DEFINED
#define kRepFrame_POINTER_DEFINED
EOS_PTR_CLASS(kRepFrame)
#endif

#ifndef MDIFRAME_HPP
#include <mdiframe.hpp>
#endif

class EOS_MODIFIER kRepFrame: public EosMDIFramework {
public:

public:
 EOS_GET_SDO(kRepFrame)
 kRepFrame();
 kRepFrame(const kRepFrame &orig);
 virtual ~kRepFrame();
 kRepFrame &operator =(const kRepFrame &orig);
// Begin Non-Interactive User Class Declaration
```

```
public:
// End Non-Interactive User Class Declaration
};

#endif // KRFRAME_HPP
```

```
// krframe.pre
```

```
// krframe.non

virtual void undo() { EosMDIFramework::undo(); }
virtual void redo() { EosMDIFramework::redo(); }
virtual void clear() {}
```

```
// krlife.cpp
// User file generated by Utah.
// All user changes should be done in this file.
// This file is generated only once, it won't be regenerated.
// The full skeleton is available in krlife.csk.
//
// Author:Hung-Chou Tai
// Place:MIT
// Last Modified:4/2/95
// Description:

#ifndef KRLIFE_HPP
#include <krlife.hpp>
#endif

#ifndef PROBECOM_HPP
#include <probecom.hpp>
#endif

kRepLifeline::kRepLifeline() : kRepLink()
{
  name="Lifeline";
}

kRepLifeline::~kRepLifeline()
{
}

kRepLifeline &kRepLifeline::operator =(const kRepLifeline &orig)
{
 EOS_PROBE_LOCK(*this);
 if (this != &orig)
 {
  kRepLink::operator=(orig);
```

```
  }
 return *this;
}

kRepLifeline::kRepLifeline(const kRepLifeline &orig) : kRepLink(orig)
{
}

kRepLifeline::kRepLifeline(const kRepLink &orig, const EosRect &frame) : kRepLink(orig)
{
  name="Lifeline";
  shape->setFrame(frame);
}
```

```
// krlife.hpp
// Header Generated by Utah, don't tread on me.
// Portions Copyright 1994 ViewSoft, Inc. All rights reserved.
#ifdef EOS_STRICT_INCLUDE
 #ifdef KRLIFE_HPP
  #error file already included
 #endif
#endif

#ifndef KRLIFE_HPP
#define KRLIFE_HPP
// User code for this class is in krlife
#ifndef EOSDEFS_HPP
#include <eosdefs.hpp>
#endif

#ifndef POINTER_HPP
#include <pointer.hpp>
#endif


///// User Pre-Declaration Code /////
#include "krlife.pre"

///// End User Pre-Declaration Code /////

#ifndef kRepLifeline_POINTER_DEFINED
#define kRepLifeline_POINTER_DEFINED
EOS_PTR_CLASS(kRepLifeline)
#endif

#ifndef KRLINK_HPP
#include <krlink.hpp>
#endif

class EOS_MODIFIER kRepLifeline: public kRepLink {
public:
```

```
public:
  EOS_GET_SDO(kRepLifeline)
  kRepLifeline();
  kRepLifeline(const kRepLifeline &orig);
  virtual ~kRepLifeline();
  kRepLifeline &operator =(const kRepLifeline &orig);
// Begin Non-Interactive User Class Declaration
public:
#include "krlife.non"

// End Non-Interactive User Class Declaration
};

#endif // KRLIFE_HPP
```

```
// krlife.pre
```

```
// krlife.non

kRepLifeline(const kRepLink &, const EosRect&);
```

```
// krline.cpp
// User file generated by Utah.
// All user changes should be done in this file.
// This file is generated only once, it won't be regenerated.
// The full skeleton is available in krline.csk.
//
// Author:Hung-Chou Tai
// Place:MIT
// Last Modified:4/2/95
// Description:

#ifndef KRLINE_HPP
#include <krline.hpp>
#endif

#ifndef PROBECOM_HPP
#include <probecom.hpp>
#endif

#include <math.h>

kRepLine::kRepLine() : kRepShape()
{
}

kRepLine::~kRepLine()
{
```

```
}

kRepLine &kRepLine::operator =(const kRepLine &orig)
{
 EOS_PROBE_LOCK(*this);
 if (this != &orig)
 {
  kRepShape::operator=(orig);
 }
 return *this;
}

kRepLine::kRepLine(const kRepLine &orig) : kRepShape(orig)
{
}

kRepLine::kRepLine(const EosRect &frame) : kRepShape(frame)
{
}

kRepLine::kRepLine(const kRepShape &orig) : kRepShape(orig)
{
}

void kRepLine::drawFeedback(kRepView &view, const EosRect &frame)
{
  kRepShape::drawFeedback(view, frame);
  view.drawLine(begin, end);
}

void kRepLine::draw(kRepView &view)
{
  kRepShape::draw(view);
  view.setLineThickness(1);
  view.setLineType(eosSolidLine);
  view.drawLine(begin, end);
}

EosBoolean kRepLine::containsPoint(const EosPoint &point)
{
  return onLine(point, begin, end);
}

EosBoolean kRepLine::onLine(EosPoint pt, EosPoint begin, EosPoint end)
{
  double angle, distance;
  EosPoint newpt;
  EosRect frame;
  // First everything to origin.
  end=end-begin;
  pt=pt-begin;
  begin=EosPoint(0,0);
  // Now rotate line so that it is parallel to x-axis and get angle.
  angle=atan2(end.y, end.x);
```

```
  distance=sqrt(end.x*end.x+end.y*end.y);
  newpt=EosPoint(pt.x*cos(angle)+pt.y*sin(angle),
          -pt.x*sin(angle)+pt.y*cos(angle));
  // Set frame equal to rectangle formed by the line translated (3,3)
  // centered within the rectangle
  frame.left=0;
  frame.top=0;
  frame.bottom=6;
  frame.right=distance+6;
  newpt=newpt+EosPoint(3,3);
  return frame.pointInRect(newpt);
}
```

```
// krline.hpp
// Header Generated by Utah, don't tread on me.
// Portions Copyright 1994 ViewSoft, Inc. All rights reserved.
#ifdef EOS_STRICT_INCLUDE
 #ifdef KRLINE_HPP
  #error file already included
 #endif
#endif

#ifndef KRLINE_HPP
#define KRLINE_HPP
// User code for this class is in krline
#ifndef EOSDEFS_HPP
#include <eosdefs.hpp>
#endif

#ifndef POINTER_HPP
#include <pointer.hpp>
#endif


///// User Pre-Declaration Code /////
#include "krline.pre"

///// End User Pre-Declaration Code /////

#ifndef kRepLine_POINTER_DEFINED
#define kRepLine_POINTER_DEFINED
EOS_PTR_CLASS(kRepLine)
#endif

#ifndef KRSHAPE_HPP
#include <krshape.hpp>
#endif

class EOS_MODIFIER kRepLine: public kRepShape {
public:

public:
```

```
  EOS_GET_SDO(kRepLine)
  kRepLine();
  kRepLine(const kRepLine &orig);
  virtual ~kRepLine();
  kRepLine &operator =(const kRepLine &orig);
// Begin Non-Interactive User Class Declaration
public:
#include "krline.non"

// End Non-Interactive User Class Declaration
};

#endif // KRLINE_HPP
```

// krline.pre

// krline.non

```
kRepLine(const EosRect &);
kRepLine(const kRepShape &);
virtual void draw(kRepView&);
virtual void drawFeedback(kRepView &, const EosRect &);
//virtual void setFrame(const EosRect &);
//virtual void setFrame(const EosRect &, const EosPoint &);
virtual EosBoolean containsPoint(const EosPoint&);
EosBoolean onLine(EosPoint, EosPoint, EosPoint);
```

```
// krlink.cpp
// User file generated by Utah.
// All user changes should be done in this file.
// This file is generated only once, it won't be regenerated.
// The full skeleton is available in krlink.csk.
//
// Author:Hung-Chou Tai
// Place:MIT
// Last Modified:4/2/95
// Description:
//    Link class member function definitions.
//    Check krlink.hpp, krlink.pre, and krlink.non
//    for declarations

#ifndef KRLINK_HPP
#include <krlink.hpp>
#endif

#ifndef PROBECOM_HPP
#include <probecom.hpp>
#endif
```

```
#include <editarg.hpp>
#include "krline.hpp"
#include "krepis.hpp"
#include "krsbepis.hpp"

#define LINKEDITVIEW "Edit episode view"

kRepLink::kRepLink() : kRepEpisode()
{
  selfControl=eosFalse;
  name="Link";
  delete (kRepShape*) shape;
  shape=new kRepLine();
}

kRepLink::~kRepLink()
{
}

kRepLink &kRepLink::operator =(const kRepLink &orig)
{
 EOS_PROBE_LOCK(*this);
 if (this != &orig)
 {
  kRepEpisode::operator=(orig);
  delete (kRepShape*) shape;
  shape=new kRepLine();
 }
 return *this;
}

kRepLink::kRepLink(const kRepLink &orig) : kRepEpisode(orig)
{
  delete (kRepShape*) shape;
  shape=new kRepLine(*orig.shape);
}

kRepLink::kRepLink(const kRepLink &orig, const EosRect &frame) :
  kRepEpisode(orig)
{
  delete (kRepShape*) shape;
  shape=new kRepLine(frame);
}

kRepLink::kRepLink(const EosRect &frame) : kRepEpisode()
{
  selfControl=eosFalse;
  delete (kRepShape*) shape;
  shape=new kRepLine(frame);
  name="Link";
}

EosBoolean kRepLink::containsPoint(const EosPoint& point)
```

```
{
  return shape->containsPoint(point);
}

EosBoolean kRepLink::removeAll(kRepEpisode *curEpis, kRepEpisodeList *lst)
{
  if(curEpis==getFromEpisode())
  {
    getToEpisode()->removeAssociations(lst, eosTrue);
    return eosTrue;
  }
  if(curEpis==getToEpisode())
  {
    getFromEpisode()->removeAssociations(lst, eosTrue);
    return eosTrue;
  }
  return eosFalse;
}

void kRepLink::setFromEpisode(kRepEpisode *from)
{
  addEpisode(from, FROMINDEX);
}

void kRepLink::setToEpisode(kRepEpisode *to)
{
  addEpisode(to, TOINDEX);
}

void kRepLink::draw(kRepView &view)
{
  EosPoint begin=shape->getBegin();
  EosPoint end=shape->getEnd();
  EosPoint mid((begin.x+end.x)/2, (begin.y+end.y)/2);

  view.setDisplayFilled(eosFalse);
  shape->draw(view);
  if(showInstance)
    view.drawText(name+": "+instance, mid, -1);
  else
    view.drawText(name, mid, -1);
}

void kRepLink::drawFeedback(kRepView &view, const EosRect &frame)
{
  shape->drawFeedback(view, frame);
}

void kRepLink::setupLink(kRepSubEpisode *from, kRepSubEpisode *to, tools id, kRepEpisodeList *lst)
{
  // Create subepisodes for this link and reference them to the
  // subepisodes setup above.
  kRepSubEpisode *linkFrom=new kRepSubEpisode(id, eosTrue, destination, "from");
  kRepSubEpisode *linkTo=new kRepSubEpisode(id, eosTrue, origination, "to");
```

```
  // Add these subepisodes to episodeList of link.
  linkFrom->addEpisode(from);
  linkTo->addEpisode(to);
  from->addEpisode(linkFrom);
  to->addEpisode(linkTo);
  //
  setFromEpisode(linkFrom);
  setToEpisode(linkTo);
  lst->addObject(linkFrom);
  lst->addObject(linkTo);
}

void kRepLink::removeAssociations(kRepEpisodeList *lst, EosBoolean del)
{
  EosIndex i, length=episodeList->getLength(), j;

  for(j=length-1; j>=0; j--)
  {
    kRepEpisode *tmp=(kRepEpisode*)episodeList->getObject(j);
    i=lst->findEpisode(tmp);
    if(i>=0)
    {
      if(j>0)
        tmp->parentIsDeletable(eosFalse);
      else
        tmp->parentIsDeletable(eosTrue);
      lst->remove(i);
      tmp->removeAssociations(lst, del);
      if(del)
        delete tmp;
      else
        tmp->parentIsDeletable(eosTrue);
    }
  }
}

EosBoolean kRepLink::isParentOf(kRepEpisode *cur)
{
  return((cur==getFromEpisode())||(cur==getToEpisode()));
}

EosBoolean kRepLink::containsEpisode(kRepEpisode *cur)
{
  return eosFalse;
}

EosBoolean kRepLink::allowSubEpisode(kRepSubEpisode *test)
{
  return eosTrue;
}

void kRepLink::edit(kRepView &view)
{
  EosAtom *editView=new EosAtom(LINKEDITVIEW);
```

```
  EosEditArg *editArg=new EosEditArg(*editView);
  editWait(*editArg);
}

void kRepLink::makePretty(EosPoint &fromP, EosPoint &toP, kRepEpisode *from, kRepEpisode *to)
{
  double slope;
  double yInt;
  kRepShape *fromShape=from->getShape();
  kRepShape *toShape=to->getShape();
  EosPoint begin=shape->getBegin();
  EosPoint end=shape->getEnd();
  EosRect fromFrame;
  EosRect toFrame;
  fromShape->getFrame(fromFrame);
  toShape->getFrame(toFrame);

  if(begin.x==end.x)
    if(begin.y>end.y)
    {
      while(fromFrame.pointInRect(begin))
        begin.y--;
//    begin.y+=SLOP;
      fromFrame.constrain(begin);
      while(toFrame.pointInRect(end))
        end.y++;
//    end.y-=SLOP;
      toFrame.constrain(end);
      fromP=begin;
      toP=end;
      shape->setBegin(begin);
      shape->setEnd(end);
      return;
    }
    else
    {
      while(fromFrame.pointInRect(begin))
        begin.y++;
//    begin.y-=SLOP;
      fromFrame.constrain(begin);
      while(toFrame.pointInRect(end))
        end.y--;
//    end.y+=SLOP;
      toFrame.constrain(end);
      fromP=begin;
      toP=end;
      shape->setBegin(begin);
      shape->setEnd(end);
      return;
    }

  if(begin.y==end.y)
    if(begin.x>end.x)
    {
```

```
        while(fromFrame.pointInRect(begin))
          begin.x--;
//      begin.x+=SLOP;
        fromFrame.constrain(begin);
        while(toFrame.pointInRect(end))
          end.x++;
//      end.x-=SLOP;
        toFrame.constrain(end);
        fromP=begin;
        toP=end;
        shape->setBegin(begin);
        shape->setEnd(end);
        return;
      }
      else
      {
        while(fromFrame.pointInRect(begin))
          begin.x++;
//      begin.x-=SLOP;
        fromFrame.constrain(begin);
        while(toFrame.pointInRect(end))
          end.x--;
//      end.x+=SLOP;
        toFrame.constrain(end);
        fromP=begin;
        toP=end;
        shape->setBegin(begin);
        shape->setEnd(end);
        return;
      }

    slope=((double)begin.y-(double)end.y)/((double)begin.x-(double)end.x);
    yInt=(double)begin.y-(slope*(double)begin.x);

    if(begin.x > end.x)
    {
      while(fromFrame.pointInRect(begin))
      {
        begin.x--;
        begin.y=(slope*(double)begin.x)+yInt;
      }
      fromFrame.constrain(begin);
//    begin.x+=SLOP;
//    begin.y=(slope*(double)begin.x)+yInt;
      while(toFrame.pointInRect(end))
      {
        end.x++;
        end.y=(slope*(double)end.x)+yInt;
      }
      toFrame.constrain(end);
//    end.x-=SLOP;
//    end.y=(slope*(double)end.x)+yInt;
      fromP=begin;
      toP=end;
```

```
    shape->setBegin(begin);
    shape->setEnd(end);
  }
  else
  {
    while(fromFrame.pointInRect(begin))
    {
      begin.x++;
      begin.y=(slope*(double)begin.x)+yInt;
    }
    fromFrame.constrain(begin);
//  begin.x-=SLOP;
//  begin.y=(slope*(double)begin.x)+yInt;
    while(toFrame.pointInRect(end))
    {
      end.x--;
      end.y=(slope*(double)end.x)+yInt;
    }
    toFrame.constrain(end);
//  end.x+=SLOP;
//  end.y=(slope*(double)end.x)+yInt;
    fromP=begin;
    toP=end;
    shape->setBegin(begin);
    shape->setEnd(end);
  }
}

void kRepLink::derive(kRepEpisode *from, kRepEpisode *to, tools id, kRepEpisodeList *lst, kRepEpisode
*creator)
{
  EosPoint begin, end;
  kRepShape *fromShape=from->getShape();
  kRepShape *toShape=to->getShape();
  EosRect fromFrame;
  EosRect toFrame;
  fromShape->getFrame(fromFrame);
  toShape->getFrame(toFrame);
  begin=EosPoint((fromFrame.left+fromFrame.right)/2, (fromFrame.top+fromFrame.bottom)/2);
  end=EosPoint((toFrame.left+toFrame.right)/2, (toFrame.top+toFrame.bottom)/2);
  shape->setBegin(begin);
  shape->setEnd(end);
  makePretty(begin, end, from, to);
  kRepSubEpisode *fromSub=new kRepSubEpisode(EosRect(begin, begin).inflate(
            EosPoint(SUBEPSIZE, SUBEPSIZE)), id,
            eosFalse, origination, "->"+name, creator);
  kRepSubEpisode *toSub=new kRepSubEpisode(EosRect(end, end).inflate(
            EosPoint(SUBEPSIZE, SUBEPSIZE)), id,
            eosFalse, destination, name+"->", creator);
  setupLink(fromSub, toSub, id, lst);
  from->addEpisode(fromSub);
  to->addEpisode(toSub);
  lst->addObject(fromSub);
  lst->addObject(toSub);
```

```
  lst->addObject(this);
}
```

```
// krlink.hpp
// Header Generated by Utah, don't tread on me.
// Portions Copyright 1994 ViewSoft, Inc. All rights reserved.
#ifdef EOS_STRICT_INCLUDE
 #ifdef KRLINK_HPP
  #error file already included
 #endif
#endif

#ifndef KRLINK_HPP
#define KRLINK_HPP
// User code for this class is in krlink
#ifndef EOSDEFS_HPP
#include <eosdefs.hpp>
#endif

#ifndef POINTER_HPP
#include <pointer.hpp>
#endif


///// User Pre-Declaration Code /////
#include "krlink.pre"

///// End User Pre-Declaration Code /////

#ifndef kRepLink_POINTER_DEFINED
#define kRepLink_POINTER_DEFINED
EOS_PTR_CLASS(kRepLink)
#endif

#ifndef KREPIS_HPP
#include <krepis.hpp>
#endif

class EOS_MODIFIER kRepLink: public kRepEpisode {
public:

public:
 EOS_GET_SDO(kRepLink)
 kRepLink();
 kRepLink(const kRepLink &orig);
 virtual ~kRepLink();
 kRepLink &operator =(const kRepLink &orig);
// Begin Non-Interactive User Class Declaration
public:
#include "krlink.non"

// End Non-Interactive User Class Declaration
```

```
};

#endif // KRLINK_HPP
```

```
// krlink.pre

#include "krsbepis.hpp"
#include "krdoc.hpp"

#define FROMINDEX 0
#define TOINDEX 1
```

```
// krlink.non

kRepLink(const EosRect&);
kRepLink(const kRepLink &, const EosRect&);

// draw functions
virtual void draw(kRepView &);
virtual void drawFeedback(kRepView &, const EosRect&);

// episode functions
virtual void setFromEpisode(kRepEpisode*);
virtual void setToEpisode(kRepEpisode*);
virtual kRepEpisode* getFromEpisode() { return (kRepEpisode*) episodeList->getObject(FROMINDEX);
}
virtual kRepEpisode* getToEpisode() { return (kRepEpisode*) episodeList->getObject(TOINDEX); }
virtual EosBoolean containsPoint(const EosPoint&);
virtual EosBoolean containsEpisode(kRepEpisode*);
virtual EosBoolean isParentOf(kRepEpisode*);

virtual EosBoolean removeAll(kRepEpisode *, kRepEpisodeList*);
virtual void removeAssociations(kRepEpisodeList*, EosBoolean);
virtual EosBoolean removeSurrounding(kRepEpisodeList*) { return eosFalse; }
virtual void setupLink(kRepSubEpisode*, kRepSubEpisode*, tools, kRepEpisodeList*);
virtual void derive(kRepEpisode*, kRepEpisode*, tools, kRepEpisodeList*, kRepEpisode *);
virtual void edit(kRepView &);
virtual void makePretty(EosPoint&, EosPoint&, kRepEpisode*, kRepEpisode*);

virtual EosBoolean allowSubEpisode(kRepSubEpisode *);
```

```
// krlinkpl.cpp
// User file generated by Utah.
// All user changes should be done in this file.
// This file is generated only once, it won't be regenerated.
// The full skeleton is available in krlinkpl.csk.
//
// Author:Hung-Chou Tai
```

```
// Place:MIT
// Last Modified:4/2/95
// Description:

#ifndef KRLINKPL_HPP
#include <krlinkpl.hpp>
#endif

#ifndef PROBECOM_HPP
#include <probecom.hpp>
#endif

kRepLinkPalette::kRepLinkPalette() : EosObject(),
 linkID(0)
{
}

kRepLinkPalette::~kRepLinkPalette()
{
}

kRepLinkPalette &kRepLinkPalette::operator =(const kRepLinkPalette &orig)
{
 EOS_PROBE_LOCK(*this);
 if (this != &orig)
 {
  EosObject::operator=(orig);
  linkID = orig.linkID;
 }
 return *this;
}

kRepLinkPalette::kRepLinkPalette(const kRepLinkPalette &orig) : EosObject(orig),
 linkID(orig.linkID)
{
}
```

```
// krlinkpl.hpp
// Header Generated by Utah, don't tread on me.
// Portions Copyright 1994 ViewSoft, Inc. All rights reserved.
#ifdef EOS_STRICT_INCLUDE
 #ifdef KRLINKPL_HPP
  #error file already included
 #endif
#endif

#ifndef KRLINKPL_HPP
#define KRLINKPL_HPP
// User code for this class is in krlinkpl
#ifndef EOSDEFS_HPP
#include <eosdefs.hpp>
#endif
```

```
#ifndef POINTER_HPP
#include <pointer.hpp>
#endif


///// User Pre-Declaration Code /////
#include "krlinkpl.pre"

///// End User Pre-Declaration Code /////

#ifndef kRepLinkPalette_POINTER_DEFINED
#define kRepLinkPalette_POINTER_DEFINED
EOS_PTR_CLASS(kRepLinkPalette)
#endif

#ifndef OBJECT_HPP
#include <object.hpp>
#endif

#ifndef EOSINT_HPP
#include <eosint.hpp>
#endif

class EOS_MODIFIER kRepLinkPalette: public EosObject {
public:

public:
  EosInt linkID;
  EOS_GET_SDO(kRepLinkPalette)
  kRepLinkPalette();
  kRepLinkPalette(const kRepLinkPalette &orig);
  virtual ~kRepLinkPalette();
  kRepLinkPalette &operator =(const kRepLinkPalette &orig);
// Begin Non-Interactive User Class Declaration
public:
#include "krlinkpl.non"

// End Non-Interactive User Class Declaration
};

#endif // KRLINKPL_HPP
```

```
// krlinkpl.pre
```

```
// krlinkpl.non

int getID() { return linkID; }
```

```
// krmsg.cpp
// User file generated by Utah.
// All user changes should be done in this file.
// This file is generated only once, it won't be regenerated.
// The full skeleton is available in krmsg.csk.

#ifndef KRMSG_HPP
#include <krmsg.hpp>
#endif

#ifndef PROBECOM_HPP
#include <probecom.hpp>
#endif

#include "krdoc.hpp"

kRepMessage::kRepMessage() : kRepEpisode(),
 message()
{
  parentDeletable=eosFalse;
  selfControl=eosFalse;
  name="Message";
}

kRepMessage::~kRepMessage()
{
}

kRepMessage &kRepMessage::operator =(const kRepMessage &orig)
{
 EOS_PROBE_LOCK(*this);
 if (this != &orig)
 {
  kRepEpisode::operator=(orig);
  message = orig.message;
 }
 return *this;
}

kRepMessage::kRepMessage(const kRepMessage &orig) : kRepEpisode(orig),
 message(orig.message)
{
}

kRepMessage::kRepMessage(kRepEpisode *originator, kRepDocument *destination, const EosString
&msg) : kRepEpisode(),
 message(msg)
{
  parentDeletable=eosFalse;
  selfControl=eosFalse;
  name="Message";
  episodeList->insertObject(originator, 0);
}
```

```
kRepMessage::kRepMessage(kRepEpisode *originator, const EosString &msg) : kRepEpisode(),
 message(msg)
{
  parentDeletable=eosFalse;
  selfControl=eosFalse;
  name="Message";
  episodeList->insertObject(originator, 0);
}

void kRepMessage::send()
{
}

void kRepMessage::send(kRepDocument *doc)
{
  doc->receiveMessage(this);
}
```

```
// krmsg.hpp
// Header Generated by Utah, don't tread on me.
// Portions Copyright 1994 ViewSoft, Inc. All rights reserved.
#ifdef EOS_STRICT_INCLUDE
 #ifdef KRMSG_HPP
  #error file already included
 #endif
#endif

#ifndef KRMSG_HPP
#define KRMSG_HPP
// User code for this class is in krmsg
#ifndef EOSDEFS_HPP
#include <eosdefs.hpp>
#endif

#ifndef POINTER_HPP
#include <pointer.hpp>
#endif


///// User Pre-Declaration Code /////
#include "krmsg.pre"

///// End User Pre-Declaration Code /////

#ifndef kRepMessage_POINTER_DEFINED
#define kRepMessage_POINTER_DEFINED
EOS_PTR_CLASS(kRepMessage)
#endif

#ifndef KREPIS_HPP
#include <krepis.hpp>
```

```
#endif

#ifndef STRING_HPP
#include <string.hpp>
#endif

class EOS_MODIFIER kRepMessage: public kRepEpisode {
public:

public:
  EosString message;
  EOS_GET_SDO(kRepMessage)
  kRepMessage();
  kRepMessage(const kRepMessage &orig);
  virtual ~kRepMessage();
  kRepMessage &operator =(const kRepMessage &orig);
// Begin Non-Interactive User Class Declaration
public:
#include "krmsg.non"

// End Non-Interactive User Class Declaration
};

#endif // KRMSG_HPP
```

```
// krmsg.pre
```

```
// krmsg.non

kRepMessage(kRepEpisode*, kRepDocument*, const EosString &);
kRepMessage(kRepEpisode*, const EosString&);
virtual void send();
virtual void send(kRepDocument*);
```

```
// krrect.cpp
// User file generated by Utah.
// All user changes should be done in this file.
// This file is generated only once, it won't be regenerated.
// The full skeleton is available in krrect.csk.
//
// Author:Hung-Chou Tai
// Place:MIT
// Last Modified:4/2/95
// Description:

#ifndef KRRECT_HPP
#include <krrect.hpp>
#endif
```

```
#ifndef PROBECOM_HPP
#include <probecom.hpp>
#endif

kRepRectangle::kRepRectangle() : kRepShape()
{
}

kRepRectangle::~kRepRectangle()
{
}

kRepRectangle &kRepRectangle::operator =(const kRepRectangle &orig)
{
 EOS_PROBE_LOCK(*this);
 if (this != &orig)
 {
  kRepShape::operator=(orig);
 }
 return *this;
}

kRepRectangle::kRepRectangle(const kRepRectangle &orig) : kRepShape(orig)
{
}

kRepRectangle::kRepRectangle(const kRepShape &orig) : kRepShape(orig)
{
}

kRepRectangle::kRepRectangle(const EosRect &frame) :
  kRepShape(frame)
{
}

void kRepRectangle::draw(kRepView &view)
{
  kRepShape::draw(view);
  EosRect frame;
  getFrame(frame);
  view.drawRect(frame);
}

void kRepRectangle::drawFeedback(kRepView &view, const EosRect &frame)
{
  kRepShape::drawFeedback(view, frame);
  view.drawRect(frame);
}
```

```
// krrect.hpp
// Header Generated by Utah, don't tread on me.
```

```
// Portions Copyright 1994 ViewSoft, Inc. All rights reserved.
#ifdef EOS_STRICT_INCLUDE
 #ifdef KRRECT_HPP
  #error file already included
 #endif
#endif

#ifndef KRRECT_HPP
#define KRRECT_HPP
// User code for this class is in krrect
#ifndef EOSDEFS_HPP
#include <eosdefs.hpp>
#endif

#ifndef POINTER_HPP
#include <pointer.hpp>
#endif


///// User Pre-Declaration Code /////
#include "krrect.pre"

///// End User Pre-Declaration Code /////

#ifndef kRepRectangle_POINTER_DEFINED
#define kRepRectangle_POINTER_DEFINED
EOS_PTR_CLASS(kRepRectangle)
#endif

#ifndef KRSHAPE_HPP
#include <krshape.hpp>
#endif

class EOS_MODIFIER kRepRectangle: public kRepShape {
public:

public:
 EOS_GET_SDO(kRepRectangle)
 kRepRectangle();
 kRepRectangle(const kRepRectangle &orig);
 virtual ~kRepRectangle();
 kRepRectangle &operator =(const kRepRectangle &orig);
// Begin Non-Interactive User Class Declaration
public:
#include "krrect.non"

// End Non-Interactive User Class Declaration
};

#endif // KRRECT_HPP
```

```
// krrect.pre
```

```
#include "krshape.hpp"
#include "krview.hpp"
#include <rect.hpp>
```

```
// krrect.non

kRepRectangle(const EosRect& frame);
kRepRectangle(const kRepShape&);

virtual void draw(kRepView &view);
virtual void drawFeedback(kRepView &view, const EosRect &frame);
```

```
// krsbepis.cpp
// User file generated by Utah.
// All user changes should be done in this file.
// This file is generated only once, it won't be regenerated.
// The full skeleton is available in krsbepis.csk.
//
// Author:Hung-Chou Tai
// Place:MIT
// Last Modified:4/2/95
// Description:
//    Subepisode class member function definitions.
//    Check krsbepis.hpp, krsbepis.pre, and krsbepis.non
//    for declarations.

#ifndef KRSBEPIS_HPP
#include <krsbepis.hpp>
#endif

#ifndef PROBECOM_HPP
#include <probecom.hpp>
#endif

#include "krdoc.hpp"
#include "krlink.hpp"
#include "krmsg.hpp"
#include "krframe.hpp"
#include "links.hpp"

extern EosFramework *mainFrame;

kRepSubEpisode::kRepSubEpisode() : kRepEpisode(),
  linkID(null),
  dir(none)
{
  parentDeletable=eosFalse;
  selfControl=eosFalse;
}
```

```
kRepSubEpisode::~kRepSubEpisode()
{
}

kRepSubEpisode &kRepSubEpisode::operator =(const kRepSubEpisode &orig)
{
 EOS_PROBE_LOCK(*this);
 if (this != &orig)
 {
  kRepEpisode::operator=(orig);
  linkID=orig.linkID;
  dir=orig.dir;
 }
 return *this;
}

kRepSubEpisode::kRepSubEpisode(const kRepSubEpisode &orig) : kRepEpisode(orig),
  linkID(null),
  dir(none)
{
}

kRepSubEpisode::kRepSubEpisode(const EosRect &frame, tools id, EosBoolean del, direction d, EosString
str, kRepEpisode *creator) : kRepEpisode(),
  linkID(id),
  dir(d)
{
  parentDeletable=del;
  selfControl=eosFalse;
  shape->setFrame(frame);
  name=str;
  creatorEpisode=creator;
}

kRepSubEpisode::kRepSubEpisode(tools id, EosBoolean del, direction d, EosString str) : kRepEpisode(),
  linkID(id),
  dir(d)
{
  parentDeletable=del;
  selfControl=eosFalse;
  name=str;
}

void kRepSubEpisode::draw(kRepView &view)
{
  if(dir==origination)
    view.setDisplayFilled(eosFalse);
  else
    view.setDisplayFilled(eosTrue);
  shape->draw(view);
}

void kRepSubEpisode::removeAssociations(kRepEpisodeList *lst, EosBoolean del)
```

```
{
  EosIndex i;

  kRepEpisode *parent=NULL;

  if(parentDeletable)
  {
    parent=findParent(lst);
    if(parent)
    {
      i=lst->findEpisode(parent);
      lst->remove(i);
      parent->removeAssociations(lst, del);
      if(del)
        delete parent;
//    return;
    }
  }
  else
  {
    parent=findParent(lst);
    if(parent&&del)
      parent->removeEpisode(this);
//  i=lst->findEpisode(this);
//  if(i>=0)
//    lst->remove(i);
  }
  // Find subepisode refered to in this subepisode...
  kRepSubEpisode *referencedSubEpisode=(kRepSubEpisode*) episodeList->getObject(0);
  i=lst->findEpisode(referencedSubEpisode);
  if(i>=0)
  {
    lst->remove(i);
    referencedSubEpisode->removeAssociations(lst, del);
    if(del)
      delete referencedSubEpisode;
  }
  // Now remove all subepisodes that were created using this one.
  for(i=lst->getLength()-1; i>=0; i--)
  {
    kRepEpisode *cur=(kRepEpisode*) lst->getObject(i);
    if(cur->getCreator()==this)
    {
      lst->remove(i);
      cur->removeAssociations(lst, del);
      i=lst->getLength()-1;
      if(del)
        delete cur;
    }
  }
}

EosBoolean kRepSubEpisode::containsEpisode(kRepEpisode *cur)
{
```

```
  return eosFalse;
}

EosBoolean kRepSubEpisode::isParentOf(kRepEpisode *cur)
{
  return eosFalse;
}


// This function is the master function that handles all the testing
// of subepisodes.  Modify this function when adding links to the system.

EosBoolean kRepSubEpisode::allowSubEpisode(kRepSubEpisode *test, kRepEpisodeList *lst)
{
  if(test==this)
    return eosTrue;
  switch(test->getLinkID())
  {
    case lifeline:
      return lifelineAllowable(test, lst);
      break;
    case contains:
      return containsAllowable(test, lst);
      break;
    case isA:
      return isAAllowable(test, lst);
      break;
    case sameAs:
      return sameAsAllowable(test, lst);
      break;
    case before:
      return beforeAllowable(test, lst);
      break;
    case after:
      return afterAllowable(test, lst);
      break;
    case notContains:
      return notContainsAllowable(test, lst);
      break;
    case notIsA:
      return notIsAAllowable(test, lst);
      break;
    case notSameAs:
      return notSameAsAllowable(test, lst);
      break;
    case notBefore:
      return notBeforeAllowable(test, lst);
      break;
    case notAfter:
      return notAfterAllowable(test, lst);
      break;
    case null:
    default:
      return eosTrue;
```

```
  }
}

EosBoolean kRepSubEpisode::lifelineAllowable(kRepSubEpisode *test, kRepEpisodeList *lst)
{
  kRepMessage *message;
  kRepFrame *workingFrame=(kRepFrame*) mainFrame;
  kRepDocument *workingDoc=(kRepDocument*) workingFrame->getCurrentDocument();
  kRepSubEpisode *ref=NULL;
  kRepEpisode *refParent=NULL;
  direction d=test->getDirection();

  switch (linkID)
  {
    case lifeline:
      if(d==dir)
      {
        if(d==destination)
          return eosTrue;
        ref=test->getTermination(lst);
        if(ref!=NULL)
        {
          refParent=ref->findParent(lst);
          if(refParent!=NULL)
            if(test->findDuplicate(refParent, lst))
            {
              message=new kRepMessage(this, "Duplicate lifeline found");
              message->send(workingDoc);
              return eosTrue;
            }
        }
      }
      else
        if(checkReflection(test, lst))
        {
          message=new kRepMessage(this, "Reflection not allowed for lifeline");
          message->send(workingDoc);
        }
      return eosTrue;
      break;
    default:
      if((test->getTermination(lst))->checkDependence(lst, test, lifeline, none))
      {
        message=new kRepMessage(this, "Lifeline not allowable with"
                        +(getReferenced()->findParent(lst))->getName());
        message->send(workingDoc);
        return eosTrue;
      }
      else
        return eosTrue;
  }
}

EosBoolean kRepSubEpisode::containsAllowable(kRepSubEpisode *test, kRepEpisodeList *lst)
```

```
{
  kRepMessage *message;
  kRepFrame *workingFrame=(kRepFrame*) mainFrame;
  kRepDocument *workingDoc=(kRepDocument*) workingFrame->getCurrentDocument();
  kRepSubEpisode *ref=NULL;
  kRepEpisode *refParent=NULL;
  direction d=test->getDirection();

  switch (linkID)
  {
    case contains:
      if(d==dir)
      {
        if(d==destination)
          return eosTrue;
        ref=test->getTermination(lst);
        if(ref!=NULL)
        {
          refParent=ref->findParent(lst);
          if(refParent!=NULL)
            if(test->findDuplicate(refParent, lst))
            {
              message=new kRepMessage(this, "Inconsistency: Duplicate contains link found");
              message->send(workingDoc);
              return eosTrue;
            }
        }
      }
      if(checkReflection(test, lst))
      {
        message=new kRepMessage(this, "Reflection not allowed for contains link");
        message->send(workingDoc);
      }
      if(dir!=d)
      {
        if((test->getTermination(lst))->checkDependence(lst, test, lifeline, none))
        {
          message=new kRepMessage(this, "Contains link not allowed with lifeline dependence");
          message->send(workingDoc);
          return eosTrue;
        }
      }
      return eosTrue;
      break;
    case lifeline:
      if(d==origination)
        return eosTrue;
      if((test->getTermination(lst))->checkDependence(lst, test, lifeline, none))
      {
        message=new kRepMessage(this, "Contains link not allowed because of lifeline");
        message->send(workingDoc);
        return eosTrue;
      }
      else
```

```
        return eosTrue;
      break;
    case notContains:
      if(d==origination)
        return eosTrue;
      if((test->getTermination(lst))->checkDependence(lst, test, notContains, dir))
      {
        message=new kRepMessage(this, "Inconsistency: Contains link not allowed
                                                because of converse relation");
        message->send(workingDoc);
        return eosTrue;
      }
      return eosTrue;
      break;
    default:
      return eosTrue;
  }
}

EosBoolean kRepSubEpisode::isAAllowable(kRepSubEpisode *test, kRepEpisodeList *lst)
{
  kRepMessage *message;
  kRepFrame *workingFrame=(kRepFrame*) mainFrame;
  kRepDocument *workingDoc=(kRepDocument*) workingFrame->getCurrentDocument();
  kRepSubEpisode *ref=NULL;
  kRepEpisode *refParent=NULL;
  direction d=test->getDirection();

  switch (linkID)
  {
    case isA:
      if(d==dir)
      {
        if(d==destination)
          return eosTrue;
        ref=test->getTermination(lst);
        if(ref!=NULL)
        {
          refParent=ref->findParent(lst);
          if(refParent!=NULL)
            if(test->findDuplicate(refParent, lst))
            {
              message=new kRepMessage(this, "Inconsistency: Duplicate is-a link found");
              message->send(workingDoc);
              return eosTrue;
            }
        }
      }
      if(checkReflection(test, lst))
      {
        message=new kRepMessage(this, "Reflection not allowed for is-a link");
        message->send(workingDoc);
        return eosTrue;
      }
```

```
        if((d!=dir)||((d==destination)&&(dir==destination)))
          return eosTrue;
        if(dir!=d)
        {
          if((test->getTermination(lst))->checkDependence(lst,
              test, lifeline, none))
          {
            message=new kRepMessage(this, "Is-a link not allowed with lifeline dependence");
            message->send(workingDoc);
            return eosTrue;
          }
          else
          {
            message=new kRepMessage(this, "Episode can only have one is-a link");
            message->send(workingDoc);
            return eosTrue;
          }
        }
        else
          return eosTrue;
        break;
      case notIsA:
        if(d==origination)
          return eosTrue;
        if((test->getTermination(lst))->checkDependence(lst, test, notIsA, d))
        {
          message=new kRepMessage(this, "Is-a link not allowed because of converse relation");
          message->send(workingDoc);
          return eosTrue;
        }
        return eosTrue;
        break;
      case lifeline:
        if(d==origination)
          return eosTrue;
        if((test->getTermination(lst))->checkDependence(lst, test, lifeline, none))
        {
          message=new kRepMessage(this, "Is-a link not allowed because of lifeline");
          message->send(workingDoc);
          return eosTrue;
        }
        else
          return eosTrue;
        break;
      default:
        return eosTrue;
        break;
    }
}

EosBoolean kRepSubEpisode::sameAsAllowable(kRepSubEpisode *test, kRepEpisodeList *lst)
{
  kRepMessage *message;
  kRepFrame *workingFrame=(kRepFrame*) mainFrame;
```

```
  kRepDocument *workingDoc=(kRepDocument*) workingFrame->getCurrentDocument();
  kRepSubEpisode *ref=NULL;
  kRepEpisode *refParent=NULL;
  direction d=test->getDirection();

  switch (linkID)
  {
    case sameAs:
      if(d==dir)
      {
        if(d==destination)
          return eosTrue;
        ref=test->getTermination(lst);
        if(ref!=NULL)
        {
          refParent=ref->findParent(lst);
          if(refParent!=NULL)
            if(test->findDuplicate(refParent, lst))
            {
              message=new kRepMessage(this, "Inconsistency: Duplicate sameAs link found");
              message->send(workingDoc);
              return eosTrue;
            }
        }
      }
      return eosTrue;
      break;
    case notSameAs:
      if(d==origination)
        return eosTrue;
      if((test->getTermination(lst))->checkDependence(lst, test, notSameAs, dir))
      {
        message=new kRepMessage(this,
          "Inconsistency: Same-as link not allowed because of converse relation");
        message->send(workingDoc);
        return eosTrue;
      }
      return eosTrue;
      break;
    default:
      return eosTrue;
  return eosTrue;
  }
}

EosBoolean kRepSubEpisode::beforeAllowable(kRepSubEpisode *test, kRepEpisodeList *lst)
{
  kRepMessage *message;
  kRepFrame *workingFrame=(kRepFrame*) mainFrame;
  kRepDocument *workingDoc=(kRepDocument*) workingFrame->getCurrentDocument();
  kRepSubEpisode *ref;
  kRepEpisode *refParent;
  direction d=test->getDirection();
```

```
  switch (linkID)
  {
    case before:
      if(d==dir)
      {
        if(d==destination)
          return eosTrue;
        ref=test->getTermination(lst);
        if(ref!=NULL)
        {
          refParent=ref->findParent(lst);
          if(refParent!=NULL)
            if(test->findDuplicate(refParent, lst))
            {
              message=new kRepMessage(this, "Inconsistency: Duplicate before link found");
              message->send(workingDoc);
              return eosTrue;
            }
        }
      }
      if(checkReflection(test, lst))
      {
        message=new kRepMessage(this, "Reflection not allowed for before link");
        message->send(workingDoc);
      }
      return eosTrue;
      break;
    case notBefore:
      if(d==origination)
        return eosTrue;
      if((test->getTermination(lst))->checkDependence(lst, test, notBefore, dir))
      {
        message=new kRepMessage(this, "Inconsistency: Before link not allowed because
                                        of converse relation");
        message->send(workingDoc);
        return eosTrue;
      }
      return eosTrue;
      break;
    default:
      return eosTrue;
  }
}

EosBoolean kRepSubEpisode::afterAllowable(kRepSubEpisode *test, kRepEpisodeList *lst)
{
  kRepMessage *message;
  kRepFrame *workingFrame=(kRepFrame*) mainFrame;
  kRepDocument *workingDoc=(kRepDocument*) workingFrame->getCurrentDocument();
  kRepSubEpisode *ref;
  kRepEpisode *refParent;
  direction d=test->getDirection();

  switch (linkID)
```

```
  {
    case after:
      if(d==dir)
      {
        if(d==destination)
          return eosTrue;
        ref=test->getTermination(lst);
        if(ref!=NULL)
        {
          refParent=ref->findParent(lst);
          if(refParent!=NULL)
            if(test->findDuplicate(refParent, lst))
            {
              message=new kRepMessage(this, "Inconsistency: Duplicate after link found");
              message->send(workingDoc);
              return eosTrue;
            }
        }
      }
      if(checkReflection(test, lst))
      {
        message=new kRepMessage(this, "Reflection not allowed for after link");
        message->send(workingDoc);
      }
      return eosTrue;
      break;
    case notAfter:
      if(d==origination)
        return eosTrue;
      if((test->getTermination(lst))->checkDependence(lst, test, notAfter, dir))
      {
        message=new kRepMessage(this, "Inconsistency: After link not allowed
                                          because of converse relation");
        message->send(workingDoc);
        return eosTrue;
      }
      return eosTrue;
      break;
    default:
      return eosTrue;
  return eosTrue;
  }
}

EosBoolean kRepSubEpisode::notContainsAllowable(kRepSubEpisode *test, kRepEpisodeList *lst)
{
  kRepMessage *message;
  kRepFrame *workingFrame=(kRepFrame*) mainFrame;
  kRepDocument *workingDoc=(kRepDocument*) workingFrame->getCurrentDocument();
  kRepSubEpisode *ref;
  kRepEpisode *refParent;
  direction d=test->getDirection();

  switch (linkID)
```

```
  {
    case notContains:
      if(d==dir)
      {
        if(d==destination)
          return eosTrue;
        ref=test->getTermination(lst);
        if(ref!=NULL)
        {
          refParent=ref->findParent(lst);
          if(refParent!=NULL)
            if(test->findDuplicate(refParent, lst))
            {
              message=new kRepMessage(this, "Inconsistency: Duplicate _contains link found");
              message->send(workingDoc);
              return eosTrue;
            }
        }
      }
      return eosTrue;
      break;
    case contains:
      if(d==origination)
        return eosTrue;
      if((test->getTermination(lst))->checkDependence(lst, test, contains, dir))
      {
        message=new kRepMessage(this, "Inconsistency: _Contains link not allowed
                                              because of converse relation");
        message->send(workingDoc);
        return eosTrue;
      }
      return eosTrue;
      break;
    default:
      return eosTrue;
  return eosTrue;
  }
}

EosBoolean kRepSubEpisode::notSameAsAllowable(kRepSubEpisode *test, kRepEpisodeList *lst)
{
  kRepMessage *message;
  kRepFrame *workingFrame=(kRepFrame*) mainFrame;
  kRepDocument *workingDoc=(kRepDocument*) workingFrame->getCurrentDocument();
  kRepSubEpisode *ref;
  kRepEpisode *refParent;
  direction d=test->getDirection();

  switch (linkID)
  {
    case notSameAs:
      if(d==dir)
      {
        if(d==destination)
```

```
            return eosTrue;
          ref=test->getTermination(lst);
          if(ref!=NULL)
          {
            refParent=ref->findParent(lst);
            if(refParent!=NULL)
              if(test->findDuplicate(refParent, lst))
              {
                message=new kRepMessage(this, "Inconsistency: Duplicate _same-as link found");
                message->send(workingDoc);
                return eosTrue;
              }
          }
        }
        return eosTrue;
        break;
      case sameAs:
        if(d==origination)
          return eosTrue;
        if((test->getTermination(lst))->checkDependence(lst, test, sameAs, dir))
        {
          message=new kRepMessage(this, "Inconsistency: _Same-as link not allowed because
                                         of converse relation");
          message->send(workingDoc);
          return eosTrue;
        }
        return eosTrue;
        break;
      default:
        return eosTrue;
    return eosTrue;
    }
}

EosBoolean kRepSubEpisode::notIsAAllowable(kRepSubEpisode *test, kRepEpisodeList *lst)
{
  kRepMessage *message;
  kRepFrame *workingFrame=(kRepFrame*) mainFrame;
  kRepDocument *workingDoc=(kRepDocument*) workingFrame->getCurrentDocument();
  kRepSubEpisode *ref;
  kRepEpisode *refParent;
  direction d=test->getDirection();

  switch (linkID)
  {
    case notIsA:
      if(d==dir)
      {
        if(d==destination)
          return eosTrue;
        ref=test->getTermination(lst);
        if(ref!=NULL)
        {
          refParent=ref->findParent(lst);
```

```
            if(refParent!=NULL)
              if(test->findDuplicate(refParent, lst))
              {
                message=new kRepMessage(this, "Inconsistency: Duplicate _is-a link found");
                message->send(workingDoc);
                return eosTrue;
              }
          }
        }
        return eosTrue;
        break;
      case isA:
        if(d==origination)
          return eosTrue;
        if((test->getTermination(lst))->checkDependence(lst, test, isA, dir))
        {
          message=new kRepMessage(this, "Inconsistency: _Is-a link not allowed because of
                                                converse relation");
          message->send(workingDoc);
          return eosTrue;
        }
        return eosTrue;
        break;
      default:
        return eosTrue;
    return eosTrue;
    }
}

EosBoolean kRepSubEpisode::notBeforeAllowable(kRepSubEpisode *test, kRepEpisodeList *lst)
{
  kRepMessage *message;
  kRepFrame *workingFrame=(kRepFrame*) mainFrame;
  kRepDocument *workingDoc=(kRepDocument*) workingFrame->getCurrentDocument();
  kRepSubEpisode *ref;
  kRepEpisode *refParent;
  direction d=test->getDirection();

  switch (linkID)
  {
    case notBefore:
      if(d==dir)
      {
        if(d==destination)
          return eosTrue;
        ref=test->getTermination(lst);
        if(ref!=NULL)
        {
          refParent=ref->findParent(lst);
          if(refParent!=NULL)
            if(test->findDuplicate(refParent, lst))
            {
              message=new kRepMessage(this, "Inconsistency: Duplicate _before link found");
              message->send(workingDoc);
```

```
                return eosTrue;
              }
            }
          }
          return eosTrue;
          break;
        case before:
          if(d==origination)
            return eosTrue;
          if((test->getTermination(lst))->checkDependence(lst, test, before, dir))
          {
            message=new kRepMessage(this, "Inconsistency: _Before link not allowed because of
                                                converse relation");
            message->send(workingDoc);
            return eosTrue;
          }
          return eosTrue;
          break;
        default:
          return eosTrue;
      return eosTrue;
      }
}

EosBoolean kRepSubEpisode::notAfterAllowable(kRepSubEpisode *test, kRepEpisodeList *lst)
{
  kRepMessage *message;
  kRepFrame *workingFrame=(kRepFrame*) mainFrame;
  kRepDocument *workingDoc=(kRepDocument*) workingFrame->getCurrentDocument();
  kRepSubEpisode *ref;
  kRepEpisode *refParent;
  direction d=test->getDirection();

  switch (linkID)
  {
    case notAfter:
      if(d==dir)
      {
        if(d==destination)
          return eosTrue;
        ref=test->getTermination(lst);
        if(ref!=NULL)
        {
          refParent=ref->findParent(lst);
          if(refParent!=NULL)
            if(test->findDuplicate(refParent, lst))
            {
              message=new kRepMessage(this, "Inconsistency: Duplicate _after link found");
              message->send(workingDoc);
              return eosTrue;
            }
        }
      }
      return eosTrue;
```

```
        break;
      case after:
        if(d==origination)
          return eosTrue;
        if((test->getTermination(lst))->checkDependence(lst, test, after, dir))
        {
          message=new kRepMessage(this, "Inconsistency: _After link not allowed because of
                                                converse relation");
          message->send(workingDoc);
          return eosTrue;
        }
        return eosTrue;
        break;
      default:
        return eosTrue;
    return eosTrue;
    }
}

kRepSubEpisode* kRepSubEpisode::getReferenced()
{
  return (kRepSubEpisode*) episodeList->getObject(0);
}

kRepSubEpisode* kRepSubEpisode::getTermination(kRepEpisodeList *lst)
{
  kRepSubEpisode *ref=getReferenced();
  kRepLink *cur=(kRepLink *) ref->findParent(lst);
  if(cur==NULL)
    return NULL;
  switch (dir)
  {
    case origination:
      return (kRepSubEpisode*) ((kRepSubEpisode*)cur->getToEpisode())->getReferenced();
    case destination:
      return (kRepSubEpisode*) ((kRepSubEpisode*)cur->getFromEpisode())->getReferenced();
    default:
      return NULL;
  }
}

EosBoolean kRepSubEpisode::checkTransitive(kRepSubEpisode *test, kRepEpisodeList *lst)
{
  kRepEpisode *parent=(test->getTermination(lst))->findParent(lst);
  kRepEpisodeList *tmpList=new kRepEpisodeList();
  kRepEpisodeList *workingList=new kRepEpisodeList();
  kRepEpisode *tmpParent=NULL;
  EosIndex i;

  tmpList->addObject(this);
  while(1)
  {
    for(i=0; i<tmpList->getLength(); i++)
    {
```

```
        kRepSubEpisode *cur=(kRepSubEpisode*) tmpList->getObject(i);
        kRepSubEpisode *ref=cur->getTermination(lst);
        if(ref)
          tmpParent=ref->findParent(lst);
        else
          break;
        if(tmpParent==parent)
          return eosTrue;
        if(tmpParent)
          tmpParent->findID(linkID, dir, workingList);
      }
      if(workingList->getLength()==0)
        return eosFalse;
      else
      {
        delete tmpList;
        tmpList=new kRepEpisodeList(*workingList);
        workingList->removeAllElements();
      }
    }
}

EosBoolean kRepSubEpisode::checkReflection(kRepSubEpisode *test, kRepEpisodeList *lst)
{
  if(test->getDirection()==origination)
    return eosFalse;
  if((test->getDirection()==destination)&&(dir==origination))
    if(checkTransitive(test, lst))
      return eosTrue;
  return eosFalse;
}

EosBoolean kRepSubEpisode::checkDependence(kRepEpisodeList *lst, kRepSubEpisode *exclude, tools
dependent, direction d)
{
  kRepEpisode *parent=findParent(lst);
  kRepEpisodeList *tmpList=new kRepEpisodeList();
  kRepEpisodeList *workingList=new kRepEpisodeList();
  kRepEpisode *tmpParent=NULL;
  EosIndex i, j;

  tmpList->addObject(this);
  while(1)
  {
    for(i=0; i<tmpList->getLength(); i++)
    {
      kRepSubEpisode *cur=(kRepSubEpisode*) tmpList->getObject(i);
      kRepSubEpisode *ref=cur->getTermination(lst);
      if(ref)
        tmpParent=ref->findParent(lst);
      else
        break;
      if(cur!=exclude)
      {
```

```
            if(tmpParent==parent)
//              if(cur->getLinkID()==dependent)
                  return eosTrue;
            if(tmpParent)
            {
// Add to working list all links associated with the same link being
// excluded and the dependent link type.
              tmpParent->findID(exclude->getLinkID(), cur->getOppositeDirection(), workingList);
              tmpParent->findID(dependent, d, workingList);
              while(1)
              {
                j=workingList->findEpisode(ref);
                if(j>=0)
                workingList->remove(j);
                else
                break;
              }
            }
          }
        }
        if(workingList->getLength()==0)
          return eosFalse;
        else
        {
          delete tmpList;
          tmpList=new kRepEpisodeList(*workingList);
          workingList->removeAllElements();
        }
      }
    }

direction kRepSubEpisode::getOppositeDirection()
{
  switch (dir)
  {
    case origination:
      return destination;
    case destination:
      return origination;
    default:
      return dir;
  }
}

tools kRepSubEpisode::getConverseLink()
{
  switch(linkID)
  {
    case sameAs:
      return notSameAs;
      break;
    case notSameAs:
      return sameAs;
      break;
```

```
    case contains:
      return notContains;
      break;
    case notContains:
      return contains;
      break;
    case before:
      return notBefore;
      break;
    case after:
      return notAfter;
      break;
    case notBefore:
      return before;
      break;
    case notAfter:
      return after;
      break;
    case isA:
      return notIsA;
      break;
    case notIsA:
      return isA;
      break;
    default:
      return linkID;
  }
}

tools kRepSubEpisode::alwaysGetNegativeLink()
{
  switch(linkID)
  {
    case sameAs:
      return notSameAs;
      break;
    case contains:
      return notContains;
      break;
    case before:
      return notBefore;
      break;
    case after:
      return notAfter;
      break;
    case isA:
      return notIsA;
      break;
    default:
      return linkID;
  }
}
```

```
void kRepSubEpisode::edit(kRepView &view)
{
}

void kRepSubEpisode::sendMessage(kRepMessage *message)
{
}

void kRepSubEpisode::receiveMessage(kRepMessage *message)
{
}

void kRepSubEpisode::deriveNew(kRepEpisodeList *lst)
{
  switch(linkID)
  {
    case contains:
      deriveTransitive(lst);
      deriveNegativeTransitive(lst);
      break;
    case before:
      deriveTransitive(lst);
      break;
    case after:
      deriveTransitive(lst);
      break;
    case isA:
      deriveTransitive(lst);
      break;
    case sameAs:
      deriveSameAs(lst);
      break;
    case notContains:
      deriveNegativeTransitive(lst);
      break;
    case notBefore:
      deriveNegativeTransitive(lst);
      break;
    case notAfter:
      deriveNegativeTransitive(lst);
      break;
    case notIsA:
      deriveNegativeTransitive(lst);
      break;
    default:
      break;
  }
}

void kRepSubEpisode::deriveTransitive(kRepEpisodeList *lst)
{
// if(dir==destination)
//    return;
  kRepEpisodeList *newlst=new kRepEpisodeList(*lst);
```

```
kRepEpisodeList *tmpList=new kRepEpisodeList();
kRepEpisodeList *workingList=new kRepEpisodeList();
kRepEpisode *parent=findParent(lst);
kRepEpisode *refParent=getTermination(lst)->findParent(lst);
kRepEpisode *tmpParent;
kRepSubEpisode *ref=NULL;
kRepLink *newLink;
EosIndex index, j;

if(parent)
{
  parent->findID(linkID, getOppositeDirection(), tmpList);
  parent->findID(sameAs, none, tmpList);
  if(tmpList->getLength()<=0)
    return;
}

index=newlst->findEpisode(this);
if(index>=0)
{
  newlst->remove(index);
  removeAssociations(newlst, eosFalse);
}

while(1)
{
  kRepEpisodeListIter iter(*tmpList);
  while(!iter.isDone())
  {
    kRepSubEpisode *cur=(kRepSubEpisode*) iter.getCurrent();
    ref=cur->getTermination(newlst);
    if(ref!=NULL)
    {
      tmpParent=ref->findParent(newlst);
      index=newlst->findEpisode(cur);
      if(index>=0)
      {
        newlst->remove(index);
        cur->removeAssociations(newlst, eosFalse);
      }
      if(tmpParent)
      {
        if(tmpParent!=refParent)
        {
          tmpParent->findID(linkID, getOppositeDirection(), workingList);
          tmpParent->findID(sameAs, none, workingList);
          initLink(&newLink, linkID);
          if(dir==origination)
            newLink->derive(tmpParent, refParent, linkID, lst, this);
          else
            newLink->derive(refParent, tmpParent, linkID, lst, this);
          j=newlst->findEpisode(tmpParent);
          if(j>=0)
          {
```

```
                newlst->remove(j);
//                tmpParent->removeAssociations(newlst, eosFalse);
              }
            }
          }
        }
        iter.next();
      }
      if(workingList->getLength()==0)
        return;
      else
      {
        delete tmpList;
        tmpList=new kRepEpisodeList(*workingList);
        workingList->removeAllElements();
      }
    }
}

void kRepSubEpisode::deriveNegativeTransitive(kRepEpisodeList *lst)
{
// if(dir==destination)
//   return;
  kRepEpisodeList *newlst=new kRepEpisodeList(*lst);
  kRepEpisodeList *tmpList=new kRepEpisodeList();
  kRepEpisodeList *workingList=new kRepEpisodeList();
  kRepEpisode *parent=findParent(lst);
  kRepEpisode *refParent=getTermination(lst)->findParent(lst);
  kRepEpisode *tmpParent;
  kRepSubEpisode *ref=NULL;
  kRepLink *newLink;
  tools negationID=getConverseLink();
  tools creatorID=alwaysGetNegativeLink();
  EosIndex index, j;

  if(parent)
  {
    parent->findID(linkID, getOppositeDirection(), tmpList);
    parent->findID(negationID, getOppositeDirection(), tmpList);
    parent->findID(sameAs, none, tmpList);
    if(tmpList->getLength()<=0)
      return;
  }

  index=newlst->findEpisode(this);
  if(index>=0)
  {
    newlst->remove(index);
    removeAssociations(newlst, eosFalse);
  }

  while(1)
  {
    kRepEpisodeListIter iter(*tmpList);
```

```
    while(!iter.isDone())
    {
      kRepSubEpisode *cur=(kRepSubEpisode*) iter.getCurrent();
      ref=cur->getTermination(newlst);
      if(ref!=NULL)
      {
        tmpParent=ref->findParent(newlst);
        index=newlst->findEpisode(cur);
        if(index>=0)
        {
          newlst->remove(index);
          cur->removeAssociations(newlst, eosFalse);
        }
        if(tmpParent)
        {
          if(tmpParent!=refParent)
          {
            tmpParent->findID(linkID, getOppositeDirection(), workingList);
            parent->findID(negationID, getOppositeDirection(), tmpList);
            tmpParent->findID(sameAs, none, workingList);
            initLink(&newLink, creatorID);
            if(cur->getLinkID()==negationID)
              if(dir==origination)
                newLink->derive(tmpParent, refParent, creatorID, lst, this);
              else
                newLink->derive(refParent, tmpParent, creatorID, lst, this);
            j=newlst->findEpisode(tmpParent);
            if(j>=0)
            {
              newlst->remove(j);
//              tmpParent->removeAssociations(newlst, eosFalse);
            }
          }
        }
      }
      iter.next();
    }
    if(workingList->getLength()==0)
      return;
    else
    {
      delete tmpList;
      tmpList=new kRepEpisodeList(*workingList);
      workingList->removeAllElements();
    }
  }
}


void kRepSubEpisode::deriveSameAs(kRepEpisodeList *lst)
{
  kRepEpisode *parent=findParent(lst), *refParent=NULL;
  kRepEpisode *termParent=(getTermination(lst))->findParent(lst);
  kRepEpisodeList *tmpList=new kRepEpisodeList();
```

```
  kRepLink *newLink;
  if(!parent)
    return;
  parent->findID(null, none, tmpList);
  kRepEpisodeListIter iter(*tmpList);
  while(!iter.isDone())
  {
    kRepSubEpisode *cur=(kRepSubEpisode*) iter.getCurrent();
    kRepSubEpisode *ref=cur->getTermination(lst);
    if(ref!=NULL)
    {
      refParent=ref->findParent(lst);
      if(refParent!=NULL)
        if(refParent!=termParent)
        {
          initLink(&newLink, cur->getLinkID());
          if(ref->getDirection()==destination)
          {
            if(!ref->findDuplicate(termParent, lst))
              newLink->derive(termParent, refParent, cur->getLinkID(), lst, this);
          }
          else
          {
            if(!ref->findDuplicate(termParent, lst))
              newLink->derive(refParent, refParent, cur->getLinkID(), lst, this);
          }
        }
    }
    iter.next();
  }
}

EosBoolean kRepSubEpisode::findDuplicate(kRepEpisode *to, kRepEpisodeList *lst)
{
  kRepEpisodeList *tmpList=new kRepEpisodeList();
  kRepEpisode *from=findParent(lst);
  from->findID(null, none, tmpList);
  kRepEpisodeListIter iter(*tmpList);
  while(!iter.isDone())
  {
    kRepSubEpisode *cur=(kRepSubEpisode*) iter.getCurrent();
    kRepSubEpisode *ref=cur->getTermination(lst);
    if(cur!=this)
      if(ref)
        if(cur->getLinkID()==linkID)
          if(cur->getDirection()==dir)
            if(to->isParentOf(ref))
              return eosTrue;
    iter.next();
  }
  return eosFalse;
}

EosBoolean kRepSubEpisode::checkConverse(kRepEpisode *to, kRepEpisodeList *lst)
```

```
{
  kRepEpisodeList *tmpList=new kRepEpisodeList();
  kRepEpisode *from=findParent(lst);
  tools t=getConverseLink();
  from->findID(null, none, tmpList);
  kRepEpisodeListIter iter(*tmpList);
  while(!iter.isDone())
  {
    kRepSubEpisode *cur=(kRepSubEpisode*) iter.getCurrent();
    kRepSubEpisode *ref=cur->getTermination(lst);
    if(cur!=this)
      if(ref)
        if(cur->getLinkID()==t)
          if(cur->getDirection()==dir)
            if(to->isParentOf(ref))
              return eosTrue;
    iter.next();
  }
  return eosFalse;
}

void kRepSubEpisode::initLink(kRepLink **link, tools id)
{
  switch (id)
  {
    case lifeline:
      *link=new kRepLifeline();
      break;
    case sameAs:
      *link=new sameAsLink();
      break;
    case notSameAs:
      *link=new notSameAsLink();
      break;
    case contains:
      *link=new containsLink();
      break;
    case notContains:
      *link=new notContainsLink();
      break;
    case before:
      *link=new beforeLink();
      break;
    case after:
      *link=new afterLink();
      break;
    case isA:
      *link=new isALink();
      break;
    case notIsA:
      *link=new notIsALink();
      break;
    case notBefore:
      *link=new notBeforeLink();
```

```
      break;
    case notAfter:
      *link=new notAfterLink();
      break;
    case null:
    default:
      *link=new kRepLink();
      break;
  }
}
```

```
// krsbepis.hpp
// Header Generated by Utah, don't tread on me.
// Portions Copyright 1994 ViewSoft, Inc. All rights reserved.
#ifdef EOS_STRICT_INCLUDE
 #ifdef KRSBEPIS_HPP
  #error file already included
 #endif
#endif

#ifndef KRSBEPIS_HPP
#define KRSBEPIS_HPP
// User code for this class is in krsbepis
#ifndef EOSDEFS_HPP
#include <eosdefs.hpp>
#endif

#ifndef POINTER_HPP
#include <pointer.hpp>
#endif


///// User Pre-Declaration Code /////
#include "krsbepis.pre"

///// End User Pre-Declaration Code /////

#ifndef kRepSubEpisode_POINTER_DEFINED
#define kRepSubEpisode_POINTER_DEFINED
EOS_PTR_CLASS(kRepSubEpisode)
#endif

#ifndef KREPIS_HPP
#include <krepis.hpp>
#endif

class EOS_MODIFIER kRepSubEpisode: public kRepEpisode {
public:

public:
 EOS_GET_SDO(kRepSubEpisode)
 kRepSubEpisode();
```

```
  kRepSubEpisode(const kRepSubEpisode &orig);
  virtual ~kRepSubEpisode();
  kRepSubEpisode &operator =(const kRepSubEpisode &orig);
// Begin Non-Interactive User Class Declaration
public:
#include "krsbepis.non"

// End Non-Interactive User Class Declaration
};

#endif // KRSBEPIS_HPP
```

```
// krsbepis.pre

#define SUBEPSIZE 3

#include "krview.hpp"

class kRepLink;
```

```
// krsbepis.non

// Subepisode data
tools linkID;
direction dir;

kRepSubEpisode(const EosRect&, tools, EosBoolean, direction, EosString, kRepEpisode *);
kRepSubEpisode(tools, EosBoolean, direction, EosString);

// Viewing functions
virtual void draw(kRepView&);

// Episode functions
virtual void removeAssociations(kRepEpisodeList*, EosBoolean);
virtual EosBoolean isParentOf(kRepEpisode*);
virtual EosBoolean allowSubEpisode(kRepSubEpisode *, kRepEpisodeList *);
EosBoolean lifelineAllowable(kRepSubEpisode *, kRepEpisodeList *);
EosBoolean containsAllowable(kRepSubEpisode *, kRepEpisodeList *);
EosBoolean isAAllowable(kRepSubEpisode *, kRepEpisodeList *);
EosBoolean sameAsAllowable(kRepSubEpisode *, kRepEpisodeList *);
EosBoolean beforeAllowable(kRepSubEpisode *, kRepEpisodeList *);
EosBoolean afterAllowable(kRepSubEpisode *, kRepEpisodeList *);
EosBoolean notContainsAllowable(kRepSubEpisode *, kRepEpisodeList *);
EosBoolean notIsAAllowable(kRepSubEpisode *, kRepEpisodeList *);
EosBoolean notSameAsAllowable(kRepSubEpisode *, kRepEpisodeList *);
EosBoolean notBeforeAllowable(kRepSubEpisode *, kRepEpisodeList *);
EosBoolean notAfterAllowable(kRepSubEpisode *, kRepEpisodeList *);
EosBoolean checkTransitive(kRepSubEpisode *, kRepEpisodeList *);
EosBoolean checkReflection(kRepSubEpisode *, kRepEpisodeList *);
EosBoolean checkDependence(kRepEpisodeList *, kRepSubEpisode *, tools, direction);
```

```
kRepSubEpisode* getReferenced();
kRepSubEpisode* getTermination(kRepEpisodeList *);
virtual void deriveNew(kRepEpisodeList *);
virtual void deriveTransitive(kRepEpisodeList *);
virtual void deriveSameAs(kRepEpisodeList *);
virtual void deriveNegativeTransitive(kRepEpisodeList *);
virtual void initLink(kRepLink **, tools);
EosBoolean findDuplicate(kRepEpisode *, kRepEpisodeList *);
EosBoolean checkConverse(kRepEpisode *, kRepEpisodeList *);

// Screen functions
virtual EosBoolean containsEpisode(kRepEpisode*);


// Functions for member access
direction getDirection() { return dir; }
direction getOppositeDirection();
tools getLinkID() { return linkID; }
tools getConverseLink();
tools alwaysGetNegativeLink();
virtual void edit(kRepView &);

// Messaging functions
virtual void sendMessage(kRepMessage *);
virtual void receiveMessage(kRepMessage *);
```

```
// krshape.cpp
// User file generated by Utah.
// All user changes should be done in this file.
// This file is generated only once, it won't be regenerated.
// The full skeleton is available in krshape.csk.
//
// Author:Hung-Chou Tai
// Place:MIT
// Last Modified:4/2/95
// Description:

#ifndef KRSHAPE_HPP
#include <krshape.hpp>
#endif

#ifndef KRSHAPE_HPP
#include <krshape.hpp>
#endif

#ifndef PROBECOM_HPP
#include <probecom.hpp>
#endif

kRepShape::kRepShape() : EosObject(),
  begin(EosPoint(0,0)),
  end(EosPoint(0,0))
```

```
{
}

kRepShape::~kRepShape()
{
}

kRepShape &kRepShape::operator =(const kRepShape &orig)
{
 EOS_PROBE_LOCK(*this);
 if (this != &orig)
 {
  EosObject::operator=(orig);
  begin=orig.begin;
  end=orig.end;
 }
 return *this;
}

kRepShape::kRepShape(const kRepShape &orig) : EosObject(orig),
  begin(orig.begin),
  end(orig.end)
{
}

void kRepShape::draw(kRepView &view)
{
  EosRect frame(begin, end);
  EosColor black(0,0,0);
  view.setFilledColor(black);
  view.setLineColor(black);
  view.setFilledColor(black);
  view.setLineHeight(0);
  view.setLineThickness(1);
  view.setLineType(eosSolidLine);
}

void kRepShape::invalidate(kRepView &view)
{
  EosRect frame;
  getFrame(frame);
  frame.inflate(EosPoint(10,10));
  view.invalidateView(frame, eosTrue);
}

void kRepShape::drawFeedback(kRepView &view, const EosRect &frame)
{
  EosColor black(0,0,0);
  view.setLineColor(black);
  view.setLineThickness(1);
  view.setDisplayFilled(eosFalse);
}

EosRect& kRepShape::getFrame()
```

```
{
  EosRect frame;
  frame.left=begin.x;
  frame.right=end.x;
  frame.top=begin.y;
  frame.bottom=end.y;
  return frame;
}

EosBoolean kRepShape::containsPoint(const EosPoint &point)
{
  EosRect frame;
  getFrame(frame);
  return frame.pointInRect(point);
}

kRepShape::kRepShape(const EosRect &frame) : EosObject(),
  begin(EosPoint(frame.left, frame.top)),
  end(EosPoint(frame.right, frame.bottom))
{
}

EosPoint kRepShape::getUpperLeft()
{
  if(begin.x < end.x)
    return begin;
  else
    return end;
}
```

```
// krshape.hpp
// Header Generated by Utah, don't tread on me.
// Portions Copyright 1994 ViewSoft, Inc. All rights reserved.
#ifdef EOS_STRICT_INCLUDE
 #ifdef KRSHAPE_HPP
  #error file already included
 #endif
#endif

#ifndef KRSHAPE_HPP
#define KRSHAPE_HPP
// User code for this class is in krshape
#ifndef EOSDEFS_HPP
#include <eosdefs.hpp>
#endif

#ifndef POINTER_HPP
#include <pointer.hpp>
#endif


///// User Pre-Declaration Code /////
```

```
#include "krshape.pre"

///// End User Pre-Declaration Code /////

#ifndef kRepShape_POINTER_DEFINED
#define kRepShape_POINTER_DEFINED
EOS_PTR_CLASS(kRepShape)
#endif

#ifndef OBJECT_HPP
#include <object.hpp>
#endif

class EOS_MODIFIER kRepShape: public EosObject {
public:

public:
  EOS_GET_SDO(kRepShape)
  kRepShape();
  kRepShape(const kRepShape &orig);
  virtual ~kRepShape();
  kRepShape &operator =(const kRepShape &orig);
// Begin Non-Interactive User Class Declaration
public:
#include "krshape.non"

// End Non-Interactive User Class Declaration
};

#endif // KRSHAPE_HPP
```

```
// krshape.pre

#include <rect.hpp>
#include <krview.hpp>
```

```
// krshape.non

EosPoint begin;
EosPoint end;

virtual void setFrame(const EosRect &frame) { begin.x=frame.left; end.x=frame.right; begin.y=frame.top;
end.y=frame.bottom; }
virtual void getFrame(EosRect &frame) { frame.left=begin.x; frame.right=end.x; frame.top=begin.y;
frame.bottom=end.y; }
EosRect& getFrame();

EosPoint getBegin() { return begin; }
EosPoint getEnd() { return end; }
virtual void setBegin(const EosPoint &pt) { begin=pt; }
```

```
virtual void setEnd(const EosPoint &pt) { end=pt; }

kRepShape(const EosRect& frame);

EosPoint getUpperLeft();

virtual void draw(kRepView &view);
virtual void drawFeedback(kRepView &view, const EosRect &frame);
virtual void invalidate(kRepView &view);
virtual EosBoolean containsPoint(const EosPoint&);
```

```
// krview.cpp
// User file generated by Utah.
// All user changes should be done in this file.
// This file is generated only once, it won't be regenerated.
// The full skeleton is available in krview.csk.
//
// Author:Hung-Chou Tai
// Place:MIT
// Last Modified:4/2/95
// Description:

#ifndef KRVIEW_HPP
#include <krview.hpp>
#endif

#ifndef PROBECOM_HPP
#include <probecom.hpp>
#endif

#include <sysevent.hpp>
#include <keycodes.hpp>
#include <eosclass.hpp>
#include <iterarrd.hpp>
#include <mousecmd.hpp>
#include <cursor.hpp>
#include <acmdinva.hpp>
#include <acmdset.hpp>
#include <atomlist.hpp>
#include <viewcmd.hpp>
#include <message.hpp>
#include <rect.hpp>
#include "krepis.hpp"
#include "kreplist.hpp"
#include "krframe.hpp"
#include "krshape.hpp"
#include "krdrawep.hpp"
#include "krlink.hpp"
#include "krdrawlk.hpp"
//#include "krdoc.hpp"

extern EosFramework *mainFrame;
```

```
kRepView::kRepView() : EosUserView(),
 commandObject(),
 episodeList(NULL),
 active(eosFalse)
{
}

kRepView::~kRepView()
{
  delete (kRepEpisodeList *) episodeList;
}

kRepView &kRepView::operator =(const kRepView &orig)
{
 EOS_PROBE_LOCK(*this);
 if (this != &orig)
 {
  EosUserView::operator=(orig);
  commandObject = orig.commandObject;
  delete (kRepEpisodeList*) episodeList;
  episodeList=new kRepEpisodeList(*orig.episodeList);
  active=orig.active;
 }
 return *this;
}

kRepView::kRepView(const kRepView &orig) : EosUserView(orig),
 commandObject(orig.commandObject),
 episodeList(new kRepEpisodeList(*orig.episodeList)),
 active(orig.active)
{
}

void kRepView::insertEpisode(EosCommandObject *arg)
{
}

void kRepView::removeEpisode(EosCommandObject *arg)
{
}

void kRepView::setEpisode(EosCommandObject *arg)
{
  EosArraySetCommand *setCommand=(EosArraySetCommand*) arg->getCommand();
  EosData foo;
  kRepEpisode *cur=(kRepEpisode*) (EosObject*) setCommand->getNewData(foo);
  cur->invalidate(*this);
}

void kRepView::selectEpisode(EosCommandObject *arg)
{
}
```

```
void kRepView::superSelectEpisode(EosCommandObject *arg)
{
}

void kRepView::removeAll(EosCommandObject *arg)
{
}

void kRepView::invalidateAll(EosCommandObject *arg)
{
  EosArrayInvalidateCommand* invalCommand=(EosArrayInvalidateCommand*) arg->getCommand();
  kRepEpisodeList* list=(kRepEpisodeList*) invalCommand->getArray();
  if(!episodeList)
    episodeList=list;
}

void kRepView::draw(const EosRect &frame)
{
  if(episodeList)
  {
    kRepEpisodeListIter iter(*episodeList);
    while(!iter.isDone())
    {
      kRepEpisode *cur=iter.getCurrent();
      cur->draw(*this);
      iter.next();
    }
    showSelection();
  }
}

EosBoolean kRepView::mouse(EosMouseInfo &info)
{
  switch(info.type)
  {
    case eosMouseDown:
      handleMouseDown(info);
      break;
    case eosMouseMotion:
      handleMouseMoved(info);
      break;
    case eosMouseDblClick:
      handleDoubleClick(info);
      break;
    default:
      break;
  }
  return eosTrue;
}

void kRepView::handleMouseMoved(const EosMouseInfo& info)
{
  kRepFrame* frame=(kRepFrame*) mainFrame;
  kRepDocument* document=(kRepDocument*) frame->getCurrentDocument();
```

```
  if(!document->selectorSelected())
    setCursor(EosCursor("crosshair"));
  else
    setCursor(EOS_ARROW);
}

void kRepView::handleMouseDown(const EosMouseInfo &info)
{
  switch(info.button)
  {
    case eosRight:
      handleMouseDownRight(info);
      break;
    case eosLeft:
      handleMouseDownLeft(info);
      break;
    default:
      break;
  }
}

void kRepView::handleMouseDownLeft(const EosMouseInfo &info)
{
  kRepFrame* frame=(kRepFrame*)mainFrame;
  kRepDocument* document=(kRepDocument*) frame->getCurrentDocument();
  kRepEpisode *clickEpisode=NULL, *newEpisode=NULL;
  EosMouseCommand* newDrawCommand=NULL;
  EosMessage *message=NULL;
  EosIndex clickIndex;

  if(document->selectorSelected())
  {
    deselectAll();
    clickEpisode=pointInEpisode(info.point);
    if(clickEpisode)
    {
      clickIndex=episodeList->findEpisode(clickEpisode);
      episodeList->select(eosTrue, clickIndex);
    }
  }
  else
  {
    newEpisode=new kRepEpisode(*document->getNewEpisode(),
      EosRect(info.point, info.point));
    newDrawCommand=new kRepDrawEpisode(newEpisode, this, info.point,
      EosSystemEvent::kDefaultButton, eosTrue, eosTrue, eosFalse);
    newDrawCommand->process();
    document->resetTools();
  }
}

void kRepView::handleMouseDownRight(const EosMouseInfo &info)
{
```

```
  kRepFrame* frame=(kRepFrame*)mainFrame;
  kRepDocument* document=(kRepDocument*) frame->getCurrentDocument();
  kRepEpisode *clickEpisode=NULL;
  kRepLink *newLink=NULL;
  tools toolID=(tools) document->getToolId();
  EosMouseCommand *newDrawCommand=NULL;

  clickEpisode=pointInEpisode(info.point);
  if(clickEpisode)
  {
    newLink=new kRepLink(EosRect(info.point, info.point));
    newDrawCommand=new kRepDrawLink(newLink, clickEpisode, this, toolID,
      info.point, EosSystemEvent::kDefaultButton,
      eosTrue, eosTrue, eosFalse);
    newDrawCommand->process();
    document->resetTools();
  }
}

kRepEpisode* kRepView::pointInEpisode(const EosPoint& point)
{
  kRepEpisode* episode=NULL;
  kRepEpisode* cur=NULL;
  kRepEpisodeListIter iter(*episodeList);
  while(!iter.isDone())
  {
    cur=iter.getCurrent();
    if(cur->containsPoint(point))
    {
      if(episode!=NULL)
      {
        if(episode->containsEpisode(cur))
          episode=cur;
      }
      else
        episode=cur;
    }
    iter.next();
  }
  return episode;
}

void kRepView::deselectAll()
{
  kRepEpisodeListIter iter(*episodeList);
  while(!iter.isDone())
  {
    kRepEpisode *cur=iter.getCurrent();
//  if(cur->isSelected())
    cur->setSelect(eosFalse, this);
    iter.next();
  }
}
```

```
void kRepView::invalidateSelection()
{
  kRepEpisodeListIter iter(*episodeList);
  while(!iter.isDone())
  {
    kRepEpisode *cur=iter.getCurrent();
    if(cur->isSelected())
      cur->invalidate(*this);
    iter.next();
  }
}

void kRepView::showSelection()
{
  kRepEpisodeListIter iter(*episodeList);
  while(!iter.isDone())
  {
    kRepEpisode *cur=iter.getCurrent();
    if(cur->isSelected())
      cur->setFilled(eosTrue, *this);
    iter.next();
  }
}

EosBoolean kRepView::key(EosKeyInfo &info)
{
  kRepFrame* frame=(kRepFrame*)mainFrame;
  kRepDocument* document=(kRepDocument*) frame->getCurrentDocument();
  kRepEpisode *curEpisode=document->getCurEpisode();
  EosViewCommand *command=NULL;
  EosIndex index;
  EosString *str=NULL;
  EosMessage *message=NULL;

  switch(info.key)
  {
    case EOS_BACK:
    case EOS_DELETE:
      if(!curEpisode)
        return eosTrue;
      str=new EosString("Delete "+curEpisode->getName()+"?");
      message=new EosMessage(0L, eosQuestion, eosYesNo, eosApplication,
        *str, "Delete?", 1);
      if(message->display()==eosMessageYes)
      {
        index=episodeList->findEpisode(curEpisode);
        episodeList->remove(index);
        curEpisode->removeAssociations(episodeList, eosTrue);
        invalidateView(EosRect(EosPoint(0,0),
          EosPoint(maxDrawableWidth(), maxDrawableHeight())), eosTrue);
        delete curEpisode;
      }
      break;
    default:
```

```
        break;
    }
    return eosTrue;
}

void kRepView::handleDoubleClick(const EosMouseInfo &info)
{
    kRepEpisode *cur=pointInEpisode(info.point);
    EosAtom *editView=NULL;
    EosEditArg *editArg=NULL;

    switch(info.button)
    {
/*  case(eosRight):
        if(cur)
        {
          cur->removeSurrounding(episodeList);
          invalidateView(EosRect(EosPoint(0,0),
            EosPoint(maxDrawableWidth(), maxDrawableHeight())), eosTrue);
        }
        break;*/
      case(eosLeft):
        if(cur)
        {
/*        editView=new EosAtom(EPISODEEDITVIEW);
          editArg=new EosEditArg(*editView);
          cur->editWait(*editArg);*/
          cur->edit(*this);
          invalidateView(EosRect(EosPoint(0,0),
            EosPoint(maxDrawableWidth(), maxDrawableHeight())), eosTrue);
        }
        break;
      default:
        break;
    }
}
```

```
// krview.hpp
// Header Generated by Utah, don't tread on me.
// Portions Copyright 1994 ViewSoft, Inc. All rights reserved.
#ifdef EOS_STRICT_INCLUDE
 #ifdef KRVIEW_HPP
  #error file already included
 #endif
#endif

#ifndef KRVIEW_HPP
#define KRVIEW_HPP
// User code for this class is in krview
#ifndef EOSDEFS_HPP
#include <eosdefs.hpp>
#endif
```

```
#ifndef POINTER_HPP
#include <pointer.hpp>
#endif


///// User Pre-Declaration Code /////
#include "krview.pre"

///// End User Pre-Declaration Code /////

#ifndef kRepView_POINTER_DEFINED
#define kRepView_POINTER_DEFINED
EOS_PTR_CLASS(kRepView)
#endif

#ifndef USRVIEW_HPP
#include <usrview.hpp>
#endif

#ifndef CMDOBJ_HPP
#include <cmdobj.hpp>
#endif

class EOS_MODIFIER kRepView: public EosUserView {
public:

  virtual void insertEpisode(EosCommandObject *arg);
  virtual void removeEpisode(EosCommandObject *arg);
  virtual void setEpisode(EosCommandObject *arg);
  virtual void selectEpisode(EosCommandObject *arg);
  virtual void superSelectEpisode(EosCommandObject *arg);
  virtual void removeAll(EosCommandObject *arg);
  virtual void invalidateAll(EosCommandObject *arg);
public:
  EosCommandObject commandObject;
  EOS_GET_SDO(kRepView)
  kRepView();
  kRepView(const kRepView &orig);
  virtual ~kRepView();
  kRepView &operator =(const kRepView &orig);
// Begin Non-Interactive User Class Declaration
public:
#include "krview.non"

// End Non-Interactive User Class Declaration
};

#endif // KRVIEW_HPP
```

```
// krview.pre
```

```
class kRepEpisode;
class kRepEpisodeList;

//#define EPISODEEDITVIEW "Edit episode view"
```

```
// krview.non

virtual void draw(const EosRect&);
virtual EosBoolean mouse(EosMouseInfo &);

virtual void handleMouseDown(const EosMouseInfo&);
virtual void handleMouseDownRight(const EosMouseInfo&);
virtual void handleMouseDownLeft(const EosMouseInfo&);
virtual void handleMouseMoved(const EosMouseInfo&);
virtual void handleDoubleClick(const EosMouseInfo&);

virtual EosBoolean key(EosKeyInfo &);
virtual void deselectAll(void);
virtual void invalidateSelection(void);
kRepEpisodeList* getEpisodeList(void) { return episodeList; }
virtual EosBoolean isActive(void) { return active; }
kRepEpisode* pointInEpisode(const EosPoint &);
void showSelection();

kRepEpisodeList *episodeList;
EosBoolean active;
```

```
// links.hpp

#ifndef LINKS_HPP
#define LINKS_HPP

#include "krlife.hpp"
#include "contain.hpp"
#include "sameas.hpp"
#include "isa.hpp"
#include "nisa.hpp"
#include "ncontain.hpp"
#include "nsameas.hpp"
#include "before.hpp"
#include "after.hpp"
#include "nbefore.hpp"
#include "nafter.hpp"

#endif
```

```
// nafter.cpp
// User file generated by Utah.
```

```
// All user changes should be done in this file.
// This file is generated only once, it won't be regenerated.
// The full skeleton is available in nafter.csk.

#ifndef NAFTER_HPP
#include <nafter.hpp>
#endif

#ifndef PROBECOM_HPP
#include <probecom.hpp>
#endif

notAfterLink::notAfterLink() : kRepLink()
{
  name="_After";
}

notAfterLink::~notAfterLink()
{
}

notAfterLink &notAfterLink::operator =(const notAfterLink &orig)
{
 EOS_PROBE_LOCK(*this);
 if (this != &orig)
 {
  kRepLink::operator=(orig);
 }
 return *this;
}

notAfterLink::notAfterLink(const notAfterLink &orig) : kRepLink(orig)
{
}

notAfterLink::notAfterLink(const kRepLink &orig, const EosRect &frame) : kRepLink(orig)
{
  name="_After";
  shape->setFrame(frame);
}

void notAfterLink::setupLink(kRepSubEpisode *from, kRepSubEpisode *to, tools id, kRepEpisodeList
*lst)
{
  kRepLink::setupLink(from, to, id, lst);
}
```

```
// nafter.hpp
// Header Generated by Utah, don't tread on me.
// Portions Copyright 1994 ViewSoft, Inc. All rights reserved.
#ifdef EOS_STRICT_INCLUDE
 #ifdef NAFTER_HPP
```

```
   #error file already included
  #endif
#endif

#ifndef NAFTER_HPP
#define NAFTER_HPP
// User code for this class is in nafter
#ifndef EOSDEFS_HPP
#include <eosdefs.hpp>
#endif

#ifndef POINTER_HPP
#include <pointer.hpp>
#endif


///// User Pre-Declaration Code /////
#include "nafter.pre"

///// End User Pre-Declaration Code /////

#ifndef notAfterLink_POINTER_DEFINED
#define notAfterLink_POINTER_DEFINED
EOS_PTR_CLASS(notAfterLink)
#endif

#ifndef KRLINK_HPP
#include <krlink.hpp>
#endif

class EOS_MODIFIER notAfterLink: public kRepLink {
public:

public:
  EOS_GET_SDO(notAfterLink)
  notAfterLink();
  notAfterLink(const notAfterLink &orig);
  virtual ~notAfterLink();
  notAfterLink &operator =(const notAfterLink &orig);
// Begin Non-Interactive User Class Declaration
public:
#include "nafter.non"


// End Non-Interactive User Class Declaration
};

#endif // NAFTER_HPP
```

// nafter.pre

```
// nafter.non

notAfterLink(const kRepLink&, const EosRect&);
virtual void setupLink(kRepSubEpisode*, kRepSubEpisode*, tools, kRepEpisodeList*);
```

```
// nbefore.cpp
// User file generated by Utah.
// All user changes should be done in this file.
// This file is generated only once, it won't be regenerated.
// The full skeleton is available in nbefore.csk.

#ifndef NBEFORE_HPP
#include <nbefore.hpp>
#endif

#ifndef PROBECOM_HPP
#include <probecom.hpp>
#endif

notBeforeLink::notBeforeLink() : kRepLink()
{
  name="_Before";
}

notBeforeLink::~notBeforeLink()
{
}

notBeforeLink &notBeforeLink::operator =(const notBeforeLink &orig)
{
 EOS_PROBE_LOCK(*this);
 if (this != &orig)
 {
  kRepLink::operator=(orig);
 }
 return *this;
}

notBeforeLink::notBeforeLink(const notBeforeLink &orig) : kRepLink(orig)
{
}

notBeforeLink::notBeforeLink(const kRepLink &orig, const EosRect &frame) : kRepLink(orig)
{
  name="_Before";
  shape->setFrame(frame);
}

void notBeforeLink::setupLink(kRepSubEpisode *from, kRepSubEpisode *to, tools id, kRepEpisodeList
*lst)
{
```

```
  kRepLink::setupLink(from, to, id, lst);
}
```

```
// nbefore.hpp
// Header Generated by Utah, don't tread on me.
// Portions Copyright 1994 ViewSoft, Inc. All rights reserved.
#ifdef EOS_STRICT_INCLUDE
 #ifdef NBEFORE_HPP
  #error file already included
 #endif
#endif

#ifndef NBEFORE_HPP
#define NBEFORE_HPP
// User code for this class is in nbefore
#ifndef EOSDEFS_HPP
#include <eosdefs.hpp>
#endif

#ifndef POINTER_HPP
#include <pointer.hpp>
#endif


///// User Pre-Declaration Code /////
#include "nbefore.pre"

///// End User Pre-Declaration Code /////

#ifndef notBeforeLink_POINTER_DEFINED
#define notBeforeLink_POINTER_DEFINED
EOS_PTR_CLASS(notBeforeLink)
#endif

#ifndef KRLINK_HPP
#include <krlink.hpp>
#endif

class EOS_MODIFIER notBeforeLink: public kRepLink {
public:

public:
 EOS_GET_SDO(notBeforeLink)
 notBeforeLink();
 notBeforeLink(const notBeforeLink &orig);
 virtual ~notBeforeLink();
 notBeforeLink &operator =(const notBeforeLink &orig);
// Begin Non-Interactive User Class Declaration
public:
#include "nbefore.non"
```

```
// End Non-Interactive User Class Declaration
};

#endif // NBEFORE_HPP
```

```
// nbefore.pre
```

```
// nbefore.non

notBeforeLink(const kRepLink&, const EosRect&);
virtual void setupLink(kRepSubEpisode*, kRepSubEpisode*, tools, kRepEpisodeList*);
```

```
// ncontain.cpp
// User file generated by Utah.
// All user changes should be done in this file.
// This file is generated only once, it won't be regenerated.
// The full skeleton is available in ncontain.csk.

#ifndef NCONTAIN_HPP
#include <ncontain.hpp>
#endif

#ifndef PROBECOM_HPP
#include <probecom.hpp>
#endif

notContainsLink::notContainsLink() : kRepLink()
{
  name="_Contains";
}

notContainsLink::~notContainsLink()
{
}

notContainsLink &notContainsLink::operator =(const notContainsLink &orig)
{
 EOS_PROBE_LOCK(*this);
 if (this != &orig)
 {
  kRepLink::operator=(orig);
 }
 return *this;
}

notContainsLink::notContainsLink(const notContainsLink &orig) : kRepLink(orig)
{
}
```

```
notContainsLink::notContainsLink(const kRepLink &orig, const EosRect &frame) : kRepLink(orig)
{
  name="_Contains";
  shape->setFrame(frame);
}

void notContainsLink::setupLink(kRepSubEpisode *from, kRepSubEpisode *to, tools id, kRepEpisodeList
*lst)
{
  kRepLink::setupLink(from, to, id, lst);
}
```

```
// ncontain.hpp
// Header Generated by Utah, don't tread on me.
// Portions Copyright 1994 ViewSoft, Inc. All rights reserved.
#ifdef EOS_STRICT_INCLUDE
 #ifdef NCONTAIN_HPP
  #error file already included
 #endif
#endif

#ifndef NCONTAIN_HPP
#define NCONTAIN_HPP
// User code for this class is in ncontain
#ifndef EOSDEFS_HPP
#include <eosdefs.hpp>
#endif

#ifndef POINTER_HPP
#include <pointer.hpp>
#endif


///// User Pre-Declaration Code /////
#include "ncontain.pre"

///// End User Pre-Declaration Code /////

#ifndef notContainsLink_POINTER_DEFINED
#define notContainsLink_POINTER_DEFINED
EOS_PTR_CLASS(notContainsLink)
#endif

#ifndef KRLINK_HPP
#include <krlink.hpp>
#endif

class EOS_MODIFIER notContainsLink: public kRepLink {
public:

public:
```

```
 EOS_GET_SDO(notContainsLink)
 notContainsLink();
 notContainsLink(const notContainsLink &orig);
 virtual ~notContainsLink();
 notContainsLink &operator =(const notContainsLink &orig);
// Begin Non-Interactive User Class Declaration
public:
#include "ncontain.non"

// End Non-Interactive User Class Declaration
};

#endif // NCONTAIN_HPP
```

// ncontain.pre

// ncontain.non

```
notContainsLink(const kRepLink&, const EosRect&);
virtual void setupLink(kRepSubEpisode*, kRepSubEpisode*, tools, kRepEpisodeList*);
```

```
// nisa.cpp
// User file generated by Utah.
// All user changes should be done in this file.
// This file is generated only once, it won't be regenerated.
// The full skeleton is available in nisa.csk.

#ifndef NISA_HPP
#include <nisa.hpp>
#endif

#ifndef PROBECOM_HPP
#include <probecom.hpp>
#endif

notIsALink::notIsALink() : kRepLink()
{
  name="_Is-A";
}

notIsALink::~notIsALink()
{
}

notIsALink &notIsALink::operator =(const notIsALink &orig)
{
 EOS_PROBE_LOCK(*this);
 if (this != &orig)
```

```
 {
  kRepLink::operator=(orig);
 }
 return *this;
}

notIsALink::notIsALink(const notIsALink &orig) : kRepLink(orig)
{
}

notIsALink::notIsALink(const kRepLink &orig, const EosRect &frame) : kRepLink(orig)
{
  name="_Is-A";
  shape->setFrame(frame);
}

void notIsALink::setupLink(kRepSubEpisode *from, kRepSubEpisode *to, tools id, kRepEpisodeList *lst)
{
  kRepLink::setupLink(from, to, id, lst);
}
```

```
// nisa.hpp
// Header Generated by Utah, don't tread on me.
// Portions Copyright 1994 ViewSoft, Inc. All rights reserved.
#ifdef EOS_STRICT_INCLUDE
 #ifdef NISA_HPP
  #error file already included
 #endif
#endif

#ifndef NISA_HPP
#define NISA_HPP
// User code for this class is in nisa
#ifndef EOSDEFS_HPP
#include <eosdefs.hpp>
#endif

#ifndef POINTER_HPP
#include <pointer.hpp>
#endif


///// User Pre-Declaration Code /////
#include "nisa.pre"

///// End User Pre-Declaration Code /////

#ifndef notIsALink_POINTER_DEFINED
#define notIsALink_POINTER_DEFINED
EOS_PTR_CLASS(notIsALink)
#endif
```

```
#ifndef KRLINK_HPP
#include <krlink.hpp>
#endif

class EOS_MODIFIER notIsALink: public kRepLink {
public:

public:
  EOS_GET_SDO(notIsALink)
  notIsALink();
  notIsALink(const notIsALink &orig);
  virtual ~notIsALink();
  notIsALink &operator =(const notIsALink &orig);
// Begin Non-Interactive User Class Declaration
public:
#include "nisa.non"

// End Non-Interactive User Class Declaration
};

#endif // NISA_HPP
```

```
// nisa.pre
```

```
// nisa.non

notIsALink(const kRepLink&, const EosRect&);
virtual void setupLink(kRepSubEpisode*, kRepSubEpisode*, tools, kRepEpisodeList*);
```

```
// nsameas.cpp
// User file generated by Utah.
// All user changes should be done in this file.
// This file is generated only once, it won't be regenerated.
// The full skeleton is available in nsameas.csk.

#ifndef NSAMEAS_HPP
#include <nsameas.hpp>
#endif

#ifndef PROBECOM_HPP
#include <probecom.hpp>
#endif

notSameAsLink::notSameAsLink() : kRepLink()
{
  name="_SameAs";
}
```

```
notSameAsLink::~notSameAsLink()
{
}

notSameAsLink &notSameAsLink::operator =(const notSameAsLink &orig)
{
 EOS_PROBE_LOCK(*this);
 if (this != &orig)
 {
  kRepLink::operator=(orig);
 }
 return *this;
}

notSameAsLink::notSameAsLink(const notSameAsLink &orig) : kRepLink(orig)
{
}

notSameAsLink::notSameAsLink(const kRepLink &orig, const EosRect &frame) : kRepLink(orig)
{
  name="_SameAs";
  shape->setFrame(frame);
}

void notSameAsLink::setupLink(kRepSubEpisode *from, kRepSubEpisode *to, tools id, kRepEpisodeList
*lst)
{
  kRepLink::setupLink(from, to, id, lst);
}
```

```
// nsameas.hpp
// Header Generated by Utah, don't tread on me.
// Portions Copyright 1994 ViewSoft, Inc. All rights reserved.
#ifdef EOS_STRICT_INCLUDE
 #ifdef NSAMEAS_HPP
  #error file already included
 #endif
#endif

#ifndef NSAMEAS_HPP
#define NSAMEAS_HPP
// User code for this class is in nsameas
#ifndef EOSDEFS_HPP
#include <eosdefs.hpp>
#endif

#ifndef POINTER_HPP
#include <pointer.hpp>
#endif


///// User Pre-Declaration Code /////
```

```
#include "nsameas.pre"

///// End User Pre-Declaration Code /////

#ifndef notSameAsLink_POINTER_DEFINED
#define notSameAsLink_POINTER_DEFINED
EOS_PTR_CLASS(notSameAsLink)
#endif

#ifndef KRLINK_HPP
#include <krlink.hpp>
#endif

class EOS_MODIFIER notSameAsLink: public kRepLink {
public:

public:
 EOS_GET_SDO(notSameAsLink)
 notSameAsLink();
 notSameAsLink(const notSameAsLink &orig);
 virtual ~notSameAsLink();
 notSameAsLink &operator =(const notSameAsLink &orig);
// Begin Non-Interactive User Class Declaration
public:
#include "nsameas.non"

// End Non-Interactive User Class Declaration
};

#endif // NSAMEAS_HPP
```

```
// nsameas.pre
```

```
// nsameas.non

notSameAsLink(const kRepLink&, const EosRect&);
virtual void setupLink(kRepSubEpisode*, kRepSubEpisode*, tools, kRepEpisodeList*);
```

```
// sameas.cpp
// User file generated by Utah.
// All user changes should be done in this file.
// This file is generated only once, it won't be regenerated.
// The full skeleton is available in sameas.csk.

#ifndef SAMEAS_HPP
#include <sameas.hpp>
#endif
```

```
#ifndef PROBECOM_HPP
#include <probecom.hpp>
#endif

sameAsLink::sameAsLink() : kRepLink()
{
  name="Same-As";
}

sameAsLink::~sameAsLink()
{
}

sameAsLink &sameAsLink::operator =(const sameAsLink &orig)
{
 EOS_PROBE_LOCK(*this);
 if (this != &orig)
 {
  kRepLink::operator=(orig);
 }
 return *this;
}

sameAsLink::sameAsLink(const sameAsLink &orig) : kRepLink(orig)
{
}

sameAsLink::sameAsLink(const kRepLink &orig, const EosRect &frame) : kRepLink(orig)
{
  name="Same-As";
  shape->setFrame(frame);
}

void sameAsLink::setupLink(kRepSubEpisode *from, kRepSubEpisode *to, tools id, kRepEpisodeList *lst)
{
  kRepLink::setupLink(from, to, id, lst);
}
```

```
// sameas.hpp
// Header Generated by Utah, don't tread on me.
// Portions Copyright 1994 ViewSoft, Inc. All rights reserved.
#ifdef EOS_STRICT_INCLUDE
 #ifdef SAMEAS_HPP
  #error file already included
 #endif
#endif

#ifndef SAMEAS_HPP
#define SAMEAS_HPP
// User code for this class is in sameas
#ifndef EOSDEFS_HPP
#include <eosdefs.hpp>
```

```
#endif

#ifndef POINTER_HPP
#include <pointer.hpp>
#endif


///// User Pre-Declaration Code /////
#include "sameas.pre"

///// End User Pre-Declaration Code /////

#ifndef sameAsLink_POINTER_DEFINED
#define sameAsLink_POINTER_DEFINED
EOS_PTR_CLASS(sameAsLink)
#endif

#ifndef KRLINK_HPP
#include <krlink.hpp>
#endif

class EOS_MODIFIER sameAsLink: public kRepLink {
public:

public:
  EOS_GET_SDO(sameAsLink)
  sameAsLink();
  sameAsLink(const sameAsLink &orig);
  virtual ~sameAsLink();
  sameAsLink &operator =(const sameAsLink &orig);
// Begin Non-Interactive User Class Declaration
public:
#include "sameas.non"

// End Non-Interactive User Class Declaration
};

#endif // SAMEAS_HPP
```

```
// sameas.pre
```

```
// sameas.non

sameAsLink(const kRepLink&, const EosRect&);
virtual void setupLink(kRepSubEpisode*, kRepSubEpisode*, tools, kRepEpisodeList*);
```

Bibliography

[Basu 93] Amit Basu, "A Knowledge Representation Model for Multiuser Knowledge-Based Systems", *IEEE Transactions on Knowledge and Data Engineering*, vol. 5, no. 2, pp 177-189, April 1993.

[Bobrow & Saraswat 94] Daniel G. Bobrow and Vijay A. Saraswat, "Mathematics on the Interface Towards Adaptive Engineered Systems", Xerox PARC

[Davis et al. 93] Randall Davis, Howard Shrobe, and Peter Szolovits, "What is a Knowledge Representation?", *AI Magazine*, 14(1), pp17-33, Spring 93.

[Hewitt & Manning 94] Carl Hewitt and Carl Manning, "Strategic Readiness and Affordability Issues Addressed by Continually Available Telecomputing Services Infrastructure (CATSI)", MIT LCS 1994.

[Landauer & Bellman 94] Christopher Landauer and Kirstie Bellman, "New Mathematical Foundations for Computer Science, Workbook Edition 1: 'Post-IMACS, Web Home Page, On-line Conference, MUD'", *http:///www.cs.umd.edu/~agrawala/newmath.html*.

[Meseguer 93] Jose Meseguer, "Formal Interoperability", Computer Science Laboratory, SRI International.

[Orfali et al. 95] Robert Orfali, Dan Harkey, and Jeri Edwards, "Intergalactic Client/Server Computing", *Byte*, pp108-122, April 1995

[Talcott 94] "Mathematical Foundations for Survivable Systems", *IMACS 1994*.