91-41

91-41

A Prototype Domain Modeling Environment

H. Gomaa, L. Kerschberg, C. Bosch, V. Sugumaran, I Tavakoli

Center for Software Systems Engineering
Department of Information and Software Systems Engineering
George Mason University
Fairfax, Virginia, 22030-4444

Abstract

This paper describes a prototype domain modeling environment, which has been developed at George Mason University to demonstrate the concepts of reuse of software requirements and software architectures. The prototype environment, which is application domain independent, is used to support the development of domain models and to generate target system specifications from them. The prototype environment consists of an integrated set of commercial-of-the-shelf software tools and custom developed software tools.

1. INTRODUCTION

At George Mason University, a project is underway to support software engineering lifecycles, methods, and environments to support software reuse at the requirements and design phases of the software lifecycle, in addition to the coding phase [Biggerstaff 87]. A reuse-oriented software lifecycle, the Evolutionary Domain Lifecycle [Gomaa89, Gomaa91], has been proposed, which is a highly iterative lifecycle that takes an application domain perspective allowing the development of families of systems. A domain analysis and modeling method has also been developed [Gomaa92a]. This paper describes the prototype domain modeling environment that has been developed to demonstrate these concepts.

2. DOMAIN MODELING

The Evolutionary Domain Life Cycle (EDLC) Model [Gomaa91] is a highly iterative software life cycle model that eliminates the traditional distinction between software development and maintenance. Furthermore, because new software systems are often outgrowths of existing ones, the EDLC model takes an application domain perspective allowing the development of families of systems [Parnas79], i.e. systems that share common characteristics.

A Domain Model is a problem-oriented architecture for the application domain that reflects the similarities and variations of the members of the domain. Given a domain model of an application domain, an individual target system (one of the members of the family) is created by tailoring the domain model given the requirements of the individual system. The concept of generating target systems from an application domain model has been adopted by various researchers [Batory89, Kang90, Pyster90, Lubars89].

A Domain Modeling method [Gomaa 92] describes the procedures and steps required to analyze the application domain and to develop a domain model for it.

In a domain model, an application domain is represented by means of multiple views, such that each view presents a different perspective on the domain. The different views are:

a) Aggregation Hierarchy. The Aggregation Hierarchy (AH) is used to decompose complex aggregate object types into less complex object types eventually leading to simple object types at the leaves of the hierarchy. Object types are kernel, required in all target systems or optional, only required in some target systems. At the upper levels of the hierarchy, complex objects represent subsystems.

b) Object communication diagrams. Objects in the real world are modelled as concurrent tasks [Jackson 83], which communicate with each other using messages. Messages between objects may be loosely coupled or tightly coupled [Gomaa 84]. The object communication diagrams (OCDs), which are hierarchically structured, show how objects communicate with each other.

c) State transition diagrams. Since each object is modeled as a sequential task, an object may be defined by means of a finite state machine, and represented by a state transition diagram, whose execution is by definition strictly sequential.

d) Generalization / Specialization Hierarchies. As the requirements of a given kernel or optional object type are changed to meet the specific needs of a target system, the object type may be specialized [Meyer 87]. The variants of a domain object type are stored in a Generalization / Specialization Hierarchy (GSH).

e) Feature / object dependencies. This view shows for each feature (domain requirement) the object types required to support the feature. In domain analysis, domain requirements are analyzed and categorized as kernel requirements, i.e. must be supported in all target systems, optional requirements (only required in some target systems), and mutually exclusive requirements. Some requirements need others as prerequisites.

3 OVERVIEW OF PROTOTYPE DOMAIN MODELING ENVIRONMENT

3.1 Objectives of Prototype

The EDLC and the domain modeling concept represent a radically different approach for software development compared to the traditional waterfall model. It was considered desirable to develop a proof-of-concept prototype to determine how feasible these concepts are.

The objectives of the prototype domain modeling environment were to demonstrate the feasibility of this approach, in particular:

a) Provide tool support for representing the multiple graphical views supported by the the domain modeling method.

b) Provide consistency checking between the multiple views.

c) Mapping the multiple views to a common underlying representation, namely an object repository.

d) Provide automated support for generating target system specifications from the domain model.

e) Be domain independent.

Since this was considered a daunting task, and limited resources were available for this purpose, it was decided to constrain the proof-of-concept experiment as follows:

a) From domain modeling, provide tool support for the domain analysis and specification of the EDLC.

b) From target system generation, provide tool support for the generation of the target system specification phase. In particular, it was viewed that this phase was a good candidate for a knowledge-based tool, as the procedures for target system generation could be expressed as rules, and these rules were well understood.

3.2 Tool Support for Prototype

Because of limited resources and the need to focus those resources on the innovative parts of the domain modeling environment, it was considered desirable to use existing software tools where possible. The following tools were selected.

a) An off-the-shelf CASE tool to support the creation of the multiple graphical views. An earlier survey of CASE tools indicated that there are several CASE tools that support the popular Structured Analysis and Real-Time Structured Analysis methods [Yourdon 89]. It was decided to have the domain modeling method use a graphical notation which is similar to that used by Real-Time Structured Analysis. However the semantic interpretation of the diagrams supported by the domain modeling method is radically different from that of Real-Time Structured Analysis. Another key requirement was that the CASE tool support an open system architecture, so that the information contained in the multiple views could be extracted and processed by custom tools developed for this project.

Interactive Development Enviroment's (IDE) Software Through Pictures (StP) was selected, as it satisfies the two requirements of providing multiple view graphical editors and has an open system architecture allowing the information in these views to be extracted and manipulated.

b) Use an existing user interface management system to support a user interface based on windows, menus and icons. NASA's TAE User Interface Management System was selected for this purpose.

c) Select an expert system shell as a basis for the Knowledge Based Requirements Elicitation tool, which is used by a target system requirements engineer to select the features to be included in a target system and hence provide the basis for tailoring the domain model to create a target system. The tool selected was NASA's CLIPS expert system shell.

d) Select an object-oriented programming environment. This includes an object-oriented programming language and support for an object repository. Although initially, using an object-oriented database management system (OODBMS) was considered, after some experimentation with the GemStone OODBMS, it was decided, for the proof-of-concept prototype, to implement the object repository

using the Eiffel object-oriented programming language rather than with GemStone.

One reason for that decision is that a proof-of-concept prototype does not require many of the capabilities offered by a full-featured OODBMS. Another reason is that Eiffel's support of multiple inheritance, provision for generic classes, and use of assertions to express preconditions, postconditions, and invariants makes it a more powerful language for data definition and manipulation.

3.3 Features of Prototype

In the prototype environment, Software Through Pictures is used to represent the multiple views of the domain model, although the multiple views are semantically interpreted according to the domain modeling method. The information in the multiple views is extracted, checked for consistency, and stored in an object repository.

A knowledge based tool is used to assist with target system requirements elicitation and generation of the target system specification. The tool, implemented in NASA's CLIPS shell, conducts a dialog with the human target system requirements engineer, prompting the engineer for target system specific information. The output of this tool is used to adapt the domain model to generate the target system specification.

The prototype environment is a domain independent environment. Thus it may be used to support the development of a domain model for any application domain that has been analyzed, and to generate target system specifications from it.

4 STRUCTURE OF PROTOTYPE DOMAIN MODELING ENVIRONMENT

4.1 Introduction

The scope of the prototype environment includes creating the domain model (Figure 1) and generating target system specifications from it (Figure 2). Tools supporting the creation of a domain model are as follows:

a) The Domain Modeling Graphical Editing tool. This tool allows the graphical editing of the multiple views of the domain model, consisting of the object type aggregation hierarchy, object communication diagrams, state transition diagrams, and object generalization/specialization hierarchies. This tool was developed by tailoring the Software through Pictures (StP) CASE tool to allow for the construction of the multiple views of the domain model.

b) Domain Model Relations Extractor. This tool extracts the information contained in the multiple views of the domain model from StP's TROLL relational data base, interprets the information semantically according to the domain modeling method, and stores it in a common underlying relational representation. The relational schema is based on the concepts of the domain modeling method and is tool independent.

c) Domain Model Consistency Checking Tool. This tool checks for consistency among the multiple views of the domain model. Consistency checking within one view is done in a).

d) Domain Object Repository Generator. This tool maps the information contained in the multiple views of the domain model to the Domain Object Repository.

e) Feature / Object Editor. This tool allows the feature (requirement) / object dependencies to be defined and stored in the domain object repository.

f) Domain Object Repository Report Generator. This tool generates reports on the contents of the Domain Object Repository.

Tools supporting the generation of target system specifications from the domain model (Figure 2) are as follows:

a) Domain Dependent Knowledge Base Extractor. This tool extracts, from the domain object repository, the domain specific knowledge contained in the multiple views of the domain model, as well as the feature / object dependencies, and maps this knowledge to the domain dependent knowledge base, which is used by the KBRET tool.

b) Knowledge-Based Requirements Elicitation Tool (KBRET). This Knowledge-Based tool carries out a dialogue with the user (target system requirements engineer) to elicit target system requirements, subject to the heuristics and constraints of the domain model, and hence selects and tailors the object types to be included in the target system.

c) Target System Specification Generator. Given the list of objects to be included in the target system, this tool automatically tailors the multiple views of the domain model to generate the multiple views of the target system specification.

4.2 Tool Support for Creating Multiple Views

As discussed in Section 3, IDE's Software through Pictures was selected to support the creation of the multiple graphical views of the domain model. The data flow editor (DFE) is used to represent the object communication view with the convention of interpreting the *processes* (bubbles) as the *object types* and the *data flows* (arcs) as *messages*. The data structure editor (DSE), which is based on Jackson's Structured Programming notation for data structure diagrams, is used to represent both the aggregation hierarchy and generalization/specialization hierarchies. The *sequence* and the *select* notation are used to represent the Is_Part_of and Is_a relationships in the aggregation hierarchy and generalization/specialization hierarchies, respectively. Finally, the state transition editor (STE) is used to represent the internal behavior of the state dependent objects.

4.2.1 Domain Model Relation Extractor Tool

Once the creation of the multiple views has been completed, the graphical information in the views must be mapped to an underlying common representation from which target systems can be generated. The StP environment uses a relational database for its underlying representation. The relational database consists of a set of predefined relations that are populated by the StP *data dictionary* routines.

As mentioned previously, the semantic interpretation of the multiple views differs

from the StP interpretation. The StP underlying relational schema was expanded by adding a new set of relations that captured the semantic interpretation of the domain model. The Domain Model Relation Extractor (DMRE) tool uses a set of scripts written in the Troll/USE query language that extract the domain information from the predefined set of relations, interpret them semantically based on the domain modeling method, and store the extracted information in the newly defined relations. The new set of relations serve as the interface between the front end graphical environment that captures a domain model and any other environment that uses the domain model information. The interface is independent of the graphical tools used for creating the multiple views. A new DMRE tool, however, must be developed if the front end graphical environment is changed.

The new relations and their attributes are as follows :

1) Nodes [node name, diagram name, index, characteristics, cardinality], which defines the domain object types.
2) Arcs [arc label, source, sink, diagram name], which defines the messages.
3) Externals [external node name], which defines the external object types.
4) Node_part_of [parent node name, child node name, child diagram name], which defines the aggregation hierarchy.
5) Is_a [parent node name, child node name], which defines generalization/specialization hierarchies.
6) Arc_part_of [parent arc label, child arc label], which defines aggregate message decomposition.
7) Decomposed [parent node name, parent diagram name, child diagram name], which defines the aggregate object types.
8) Diagrams [diagram name], which defines the object communication diagrams.

4.2.2 Domain Model Consistency Checker Tool

The graphical views as presented by the OCDs, the AH, the GSHs, and the STDs, each focus on one aspect of the domain being modeled. Each view looks at the domain from a different perspective. Without any automated tool, it is easy to develop views of a domain that are semantically inconsistent.

Although StP provides consistency checking routines within one particular view, there is, in general, no consistency checking among multiple views. The Domain Model Consistency Checker (DMCC) uses a set of scripts written in the Troll/USE query language that check the underlying relations for inconsistencies based on the set of rules mentioned below:

> 1- There should be a one-to-one correspondence between the object types in the ith level OCD and the object types in the corresponding level in the AH. The domain object type in the context diagram, in particular, should correspond to the root object type (top level) in the AH.
>
> 2- The root node in each GSH should correspond to a leaf node in the AH. This is due to the fact that each GSH serves as the specialization of a leaf object type in the AH.

3- For each active leaf object type in the OCDs, where active denotes a concurrent process, there must exist a state transition diagram that captures the internal behavior of the object type. Conversely, each state transition diagram must correspond to an active leaf object type in the OCDs.

4- The events in each state transition diagram (STD) should correspond to the incoming messages of the object type in the OCD that the STD is describing. The actions in each STD, on the other hand, should correspond to the outgoing messages of the same object type in the OCD.

The domain modeler runs DMCC when the graphical views are completed. DMCC performs the consistency checking between the multiple views and then displays messages describing any inconsistencies. It is then up to the domain modeler to remove the sources of inconsistencies. A domain model is considered consistent if no inconsistency is detected by this tool.

4.3 Object Repository

A key component allowing the integration and interoperation of various tools in the prototype domain modeling environment is the *object repository*. This repository is a single complex object representing a domain model; it is composed of other objects representing domain object types, features, and the relationships among them which serve to define a domain model.

Figure 3 shows the position of the object repository within the prototype environment's overall architecture. It underlies the custom-developed tools of this environment, providing them with a common set of services for accessing and manipulating information during the domain modeling process. The sections to follow describe the object repository as it has been implemented using the Eiffel object-oriented language and reuse libraries by first describing its overall *schema* and then detailing the *services* it provides to the custom-developed tools of the prototype environment.

4.3.1 Schema

Figure 4 presents a structural diagram of the object repository showing the classes of objects from which it is composed and the relationships among them. In this diagram, thin arcs represent the *inheritance* relationship (pointing from a descendant class to its parent class) and thick arcs represent the *client* relationship (pointing from a client class to its supplier class). These classes can be grouped into three categories:

1. Classes describing generic graph structures such as directed graphs, directed multigraphs, directed acyclic graphs, and trees.

2. Specializations of the above classes representing specific graph formalisms such as aggregation hierarchies, generalization/ specialization hierarchies, object communication diagrams, state transition diagrams, and feature dependency graphs.

3. Other classes necessary to support definition of the specific graph formalisms and the domain modeling method.

The paragraphs below describe each of the classes within these three categories.

Generic graph structures. Seven of the classes appearing in Figure 4 describe objects used to represent generic graph structures such as directed graphs and trees. Class *NODE* describes the nodes that exist in various forms of directed graphs. It is a client of class *ARC* because each node has associated with it an *arc set in* and an *arc set out*. Class *ARC* describes the arcs that exist in various forms of directed graphs. It is a client of class *NODE* because each arc has associated with it a *source node* and a *sink node*. Class *DAG* describes directed acyclic graphs, a form of directed graph in which no cycles may exist and in which only one arc may exist between a source node and a sink node. It is a client of classes *NODE* and *ARC* since directed graphs are defined as sets of nodes and arcs. Class *DG* is a specialization of class *DAG* describing directed graphs which allow cycles but which permit only one arc to exist between a source node and a sink node. Class *DMG* is a specialization of class *DG* describing directed multigraphs which permit more than one arc going from a single source node to a single sink node. Finally, class *TREE* describes hierarchical structures.

Specific graph formalisms. Five of the classes in Figure 4 describe the specific graph formalisms employed by the domain modeling method discussed in section 2. Classes *AH* and *GSH* are specializations of class *TREE* describing aggregation hierarchies and generalization/specialization hierarchies respectively. In both aggregation and generalization/specialization hierarchies, the roots of trees and their subtrees will reference domain object types. Class *OCD* is a specialization of class *DMG* describing object communication diagrams. Each node in an object communication diagram will reference a domain object type, and arcs will be labeled with the messages by which domain object types communicate. Class *STD*, also a specialization of class *DMG*, describes state transition diagrams. Each node in a state transition diagram will reference a state, and arcs will be labeled with transitions. Class *FDG* describes feature dependency graphs. It is a specialization of class *DAG* because the domain modeling method does not allow for circular dependencies among system features (functional capabilities). Each node in a feature dependency graph will reference a system feature, and arcs will be unlabeled.

Other classes. The remaining classes in Figure 4 are necessary to support definition of the specific graph formalisms and the domain modeling method. Class *OBJECT_TYPE* describes the definitions of domain object types, and class *SYSTEM_FEATURE* describes the features (functional capabilities) of target systems to be derived from a domain model. Class *SYSTEM_FEATURE* is a client of class *OBJECT_TYPE* because each feature may have one or more supporting domain object types required to implement that feature. Classes *STATE* and *TRANSITION* describe the states and transitions referenced by instances of class *STD*, and class *MESSAGE* describes the messages by which domain object types may communicate. Finally, class *DOMAIN_MODEL* either directly or indirectly aggregates all other classes forming the schema of the object repository, permitting the persistent storage of a single complex object representing a domain model.

4.3.2 Services

Each of the classes depicted in Figure 4 and described in the preceding paragraphs provides a set of services which allow client classes to access and manipulate the state of objects in the object repository. It is this common set of services which allows the integration and interoperation of the custom-developed tools in the environment.

Class *DAG* exports to its clients three attributes, five procedures, and eight functions. These services defined for class *DAG* are inherited and in some cases redefined by classes *DG* and *DMG*. Classes *FDG*, *STD*, *OCD*, *AGH*, and *GSH* describing specific graph formalisms define additional attributes, procedures, and functions as appropriate to those inherited from their parent classes describing generic graphs and trees. For example, the EDLC domain modeling method provides a decomposition relationship between object comunication diagrams where a specific node in one OCD can be decomposed into a set of nodes that will be shown in another OCD. To capture this relationship in the object repository, class *OCD* has attributes referencing the object communication diagram's parent diagram as well as its parent node within that diagram. In addition, class *OCD* has a function which computes the level of a particular diagram.

Services provided by the classes forming the object repository's schema such as those detailed above allow the integration and interoperation of the custom-developed tools in the prototype environment. The next section of this paper, describes some of those custom-developed tools and shows how they themselves have been integrated with other tools.

4.4 Custom-developed Tools

Within the prototype domain modeling environment, a number of custom-developed tools interact with the object repository as shown previously in Figure 3. Like the object repository itself, these tools were developed using the Eiffel object-oriented language and reuse libraries. Each of these tools is a client of class *DOMAIN_MODEL*; they make use of the services provided by this class and by other classes forming the object repository's schema. So that these tools would have a common user interface, Eiffel was used to encapsulate the TAE Plus user interface development and management system [Szczur 90]. So that some of these tools could make use of the inferencing capabilities of an expert system shell, Eiffel was used to encapsulate the C-Language Integrated Production System (CLIPS). Encapsulation of these existing tools is discussed in section 4.4.1. Section 4.4.2 then describes some of the custom-developed tools in the prototype domain modeling environment.

4.4.1 Encapsulation of Existing Tools

The TAE Plus user interface development and management system provides three packages of 'C' routines that may be called from any application which is to have an interface developed using the TAE WorkBench. These packages are as follows:

- o The Collection (Co) package

- o The Variable Manipulation (Vm) Package

o The Window Programming Tools (Wpt) package

Using Eiffel's mechanism for referencing external 'C' routines from within Eiffel routines, each of these packages has been encapsulated in its own Eiffel class and their features have been exported to the class *TAE_CLIENT*. Any custom-developed tool in the prototype environment that is to interact with the user through an interface developed using TAE's WorkBench can access the services provided by TAE's three packages of 'C' routines by inheriting from class *TAE_CLIENT*.

In addition to encapsulating the TAE Plus user interface management system, Eiffel was also used to encapsulate CLIPS expert system shell. The Eiffel language provides no rule-based processing of information, but does provide a mechanism for referencing external routines written in the 'C' language as already mentioned. Thus, it was a simple matter to encapsulate the 'C' functions provided by CLIPS within an Eiffel class; any custom-developed tool requiring rule-based processing of information can access the services provided by CLIPS by inheriting from class *CLIPS*.

4.4.2 Description of Custom-developed Tools

With TAE Plus and CLIPS encapsulated within Eiffel classes, it was possible to pursue the development of custom tools for the prototype domain modeling environment. The paragraphs below described four of the custom-developed tools in this environment.

Domain Object Repository Generator. The tool which creates the object repository is called the Domain Object Repository Generator. First it creates a single instance of class *DOMAIN_MODEL* as described in section 4.3.1. It then takes the information exported from StP in a relational representation, creates corresponding objects according to the object repository's schema, and adds those objects to the domain model. For example, if the domain analyst had created eight object communication diagrams using StP, the DIAGRAMS relation shown in Figure 5 would contain eight tuples each containing a unique string corresponding to the name of an OCD. In processing this relation, the Domain Object Repository Generator would create eight instances of class *OCD* as described in section 4.3.1 of this paper and would then add them to the instance of class *DOMAIN_MODEL* using services provided by that class.

Next, the Domain Object Repository Generator would process the NODES relation shown in Figure 5, then the ARCS relation and so on. In processing these relations, the Domain Object Repository Generator will create instances of most of the classes introduced in section 4.3.1 and will then add them to the single instance of class *DOMAIN_MODEL* using services provided by that class. The end result of processing these relations will be a single complex object representing a domain model.

Feature/Object Editor. Another tool is the Feature/Object Editor. Once the object repository representing a domain model has been created, the domain analyst can use this tool to define new features by: 1) giving each new feature a unique name, 2) entering an informal annotation for each new feature, 3) specifying domain object types supporting the feature being defined, and 4) specifying other features required by the feature being defined. In addition to defining new features, the domain analyst may use this tool to browse features previously defined for a given

domain model, delete features, or modify the definition of features in a domain model.

The Feature/Object Editor can also be used to establish relationships among sets of features. For a given set of features one of the following three types of relationships may be specified:

o no more than one feature from the set may be selected for inclusion in the target system,

o exactly one feature from the set must be selected for inclusion in the target system, or

o at least one feature from the set must be selected for inclusion in the target system.

Each set of features defined in this manner is uniquely named and may have an informal annotation associated with it. As with individual features, it is possible to use this tool to browse sets of features previously defined, delete sets of features, or modify the definitions of sets of features in a domain model.

Domain Object Repository Report Generator. Another tool which interacts with the object repository is the Domain Object Repository Report Generator tool. At any time after creating the object repository, the domain analyst can generate a report on the contents of a given domain model by using this tool which extracts select information from the object repository as specified by the domain analyst, formats that information using the LaTeX typesetting program, and displays the resulting document in an X-Windows document previewer.

Domain Dependent Knowledge Base Extractor. A final tool which interacts with the object repository is the Domain Dependent Knowledge Base Extractor. This tool extracts information about the domain model from the object repository, formats that information as CLIPS facts which can be processed by the Knowledge-Based Requirements Elicitation Tool (KBRET), and stores those facts in a file for use by that tool.

For example, if the object repository contains 100 instances of class *OBJECT_TYPE*, then the domain dependent knowledge base will contain 100 facts of the form shown below.

(Object: *id name properties*)

The *id* term in these facts is an integer uniquely identifying an object type. The *name* term in these facts is a string of characters representing the name of the object type -- this string will also be unique. The *properties* term is a sequence of strings separated blank characters. Possible properties for an object type are:

o kernel

o optional

o aggregate

o variant

o agh_root

o gsh_root

Similarly, if the object repository contains 15 instances of class *SYSTEM_FEATURE*, then the domain dependent knowledge base will contain 15 facts of the form shown below.

(Feature: *id* *name*)

The *id* term in these facts is a unique integer identifying the feature. The *name* term in these facts is a string of characters representing the name of the target system feature --- this string will also be unique.

Other facts in the domain dependent knowledge base represent the dependencies among features, the dependencies between features and domain object types, the aggregation and generalization/specialization relationships among domain object types, and so on.

4.5 Generation of Target System Specification

In addition to the multiple views, the EDLC domain model captures the reusable domain features (requirements) and the dependencies among features and object types. A target system specification is derived from the domain model by tailoring the domain model according to the features desired in the target system. During specification generation, the feature object dependencies have to be enforced in order to generate a consistent specification. The process of generating target system specifications from a domain model requires knowledge-based tool support. This tool must not only have knowledge about the domain model, but also contain procedural knowledge about constructing target systems. A knowledge-based system called the Knowledge-Based Requirements Elicitation Tool (KBRET) has been developed to automate the process of generating the specifications for the target systems. This tool has been implemented in NASA's CLIPS expert system shell. The architecture of KBRET and its components are discussed in the following sections.

4.5.1 Knowledge-Based Requirements Elicitation Tool (KBRET)

The major components of KBRET are 1) the domain dependent knowledge base, 2) the domain independent knowledge base and 3) the inference engine. The domain dependent knowledge base is derived from the object repository through the KBRET-Object repository interface and contains domain specific information. The domain independent knowledge base contains the procedural and control knowledge required in generating target system specifications from the domain model. This separation between the domain-independent and domain-dependent knowledge is essential for providing scale-up and maintainability. Also, the domain independent knowledge base can be applied to different domain models regardless of their application domain.

KBRET accomplishes the task of generating the target system specification in several phases. Some of the phases that KBRET may go through are: Browsing, Target System Requirements Elicitation, Dependency Checking, and Target System Specification Generation. The various components of KBRET are schematically

shown in Figure 6. The domain-dependent and domain-independent knowledge bases are discussed in the following sections.

4.5.1.1 Domain Dependent Knowledge Base

As the name suggests, the domain dependent knowledge base contains specific information about a particular application domain. This knowledge base is composed of several knowledge sources, namely, "Feature and Object Types", "Inter-Feature & Feature-Object Dependencies" and "Multiple Views". They are used by the domain independent knowledge base of KBRET in eliciting the requirements and generating the target system specification. The domain dependent knowledge base is derived from the domain specification, which is stored persistently in the object repository. The following paragraphs describe the knowledge sources of the domain dependent knowledge base.

The *Features and Object Types* knowledge source contains a list of all the object types and features that have been incorporated in the domain model. For each object type, its name and properties are stored in this knowledge source. The properties of objects are: kernel, optional, variant, aggregate, agh_root and gsh_root. The various relationships and dependencies among features and between features and object types are captured in the *Inter-Feature & Feature-Object Dependencies* knowledge source.

The *Multiple Views* knowledge source contains the different views created using the EDLC methodology, in particular, the aggregation hierarchy and the generalization/specialization hierarchies. These hierarchies are accessed and utilized by the *Target System Generator* knowledge source when the target system is being assembled. The following section discusses the domain independent knowledge base of KBRET.

4.5.1.2 Domain Independent Knowledge Base

The domain independent knowledge base provides procedural and control knowledge for the various functions supported by KBRET. The *User Interface Manager* is responsible for carrying out a meaningful dialog with the target system engineer to elicit the requirements for the target system. It addresses such issues as how, and in what sequence the target system engineer should be prompted for various features, invoking and controlling the different phases of KBRET.

Before specifying the requirements for the target system, the target system engineer may wish to browse through portions of the domain model in order to gain understanding of the application domain under consideration. The *Domain Browser* knowledge source provides this facility. It provides rules for initiating and terminating the browsing facility and also the appropriate domain dependent knowledge sources to be accessed in order to facilitate the browsing of those parts of the domain model which the target system engineer wishes to explore.

The *Feature & Object Selection/Deletion* knowledge source keeps track of the selection or deletion of features for the target system and the corresponding object types. This knowledge source incorporates rules for selecting and deleting features and also invoking the appropriate rules for checking inter-feature and feature/object dependencies.

The *Dependency Checker* knowledge source cooperatively works with the Feature & Object Selection/Deletion knowledge source. Whenever a feature is selected or deleted, the *Dependency Checker* enforces the inter-feature and feature/object dependencies, which are obtained from the *Inter-Feature & Feature-Object Dependencies* knowledge source. When a feature with some prerequisite features is selected, the *Dependency Checker* ensures that those prerequisite features are included in the target system. Similarly, before deleting a feature from the target system, dependency checking is performed to ensure that it is not required by any other target system feature.

Once feature selection for the target system has been completed, the *Target System Generator* knowledge source begins the process of assembling the target system. The domain kernel object types are automatically included in the target system. Depending upon the features selected for the target system, the corresponding variant and optional object types are included according to the feature/object dependencies. When the target system assembly is complete, KBRET produces two relations, one of which contains the object types that have been included in the target system and the other contains the specializations that have been included in the target system. These two relations are used in tailoring the picture files of the domain model to create the target system picture files to be displayed using StP.

On completion of the generation of a target system specification, it is stored in the object repository. The target system specification can be reused in that not only the specification is stored, but also the features and the reasoning "state" so that KBRET can be used to make "incremental changes" to an existing target system and generate a new target system specification rather than starting initially from the domain model.

4.5.2 Target System Specification Generator Tool

The graphical views of a target system can be automatically generated from those of the domain model by tailoring the domain model views based on the target system specification elicited by the Knowledge Based Requirements Elicitation Tool (KBRET). The specification of a target system is defined in terms of the object types that are to be included in the target system. Using this information, the Target System Specification Generator (TSSG) tool performs the following tasks:

1- Derives the set of object types that are not included in the target system and hence must be removed from the domain model views.

2- Generates the graphical views for the target system using the domain model views and the list of the object types to be deleted.

3- Modifies the object type names by appending the word Variant to the name of those object types for which a variant, i.e. a specialization, has been selected and the word Variants to those object types for which more than one variant object type has been selected.

5 CONCLUSION

This paper has described a prototype domain modeling environment, which has been developed at George Mason University to demonstrate the concepts of reuse

of software requirements and software architectures. The prototype environment, which is application domain independent, is used to support the development of domain models and to generate target system specifications from them. The prototype environment consists of an integrated set of commercial-of-the-shelf software tools and custom developed software tools.

6 ACKNOWLEDGEMENTS

We gratefully acknowledge the assistance of S. Bailin, R. Dutilly, J.M. Moore, and W. Truszkowski in providing us with information on the POCC. We gratefully acknowledge the major contributions of Liz O'Hara-Schettino in developing the domain model of the POCC. This work was sponsored primarily by NASA Goddard Space Flight Center, with support from the Virginia Center of Innovative Technology. The Software Through Pictures CASE tool was donated to GMU by Interactive Development Environments.

7 REFERENCES

[Batory89] "The Genesis Database System Compiler: A Result of Domain Modeling", Proc. Workshop on Domain Modeling for Software Engineering, OOPSLA'89, New Orleans, October 1989.

[Biggerstaff 87] Biggerstaff T. and Richter C., "Reusability Framework, Assessment, and Directions", IEEE Software, March 1987.

[CLIPS 89] Artificial Intelligence Section, Lyndon B. Johnson Space Center, "CLIPS Reference Manual", May 1989.

[Gomaa 84] Gomaa H, "A Software Design Method for Real Time Systems", Communications ACM, September 1984.

[Gomaa 89] Gomaa H, R Fairley and L Kerschberg, "Towards an Evolutionary Domain Life Cycle Model", Proc. Workshop on Domain Modeling for Software Engineering, OOPSLA, New Orleans, October 3, 1989.

[Gomaa 91] Gomaa H and L Kerschberg, "An Evolutionary Domain Life Cycle Model for Domain Modeling and Target System Generation", Proc. Workshop on Domain Modeling for Software Engineering, International Conference on Software Engineering, Austin, May 1991.

[Gomaa 92a] Gomaa H, "An Object-Oriented Domain Analysis and Modeling Method for Software Reuse" Proc. Hawaii International Conference on System Sciences", Hawaii, January 1992.

[Gomaa 92b] Gomaa H, L. Kerschberg, V. Sugumaran, "A Knowledge-Based Approach for Generating Target System Specifications from a Domain Model", Accepted for publication in Proc. IFIP World Computer Congress, Madrid, Spain, September 1992.

[Jackson 83] Jackson M, "System Development", Prentice Hall, 1983.

[Kang 90], Kang K.C. et. al., "Feature-Oriented Domain Analysis", Technical Report No. CMU/SEI-90-TR-21, Software Engineering Institute, November 1990.

[Lubars 89] Lubars M.D., "Domain Analysis for Multiple Target Systems", Proc. Workshop on Domain Modeling for Software Engineering, OOPSLA'89, New Orleans, October 1989.

[Meyer 87] Meyer B, "Reusability: The Case for Object-Oriented Design", IEEE Software, March 1987.

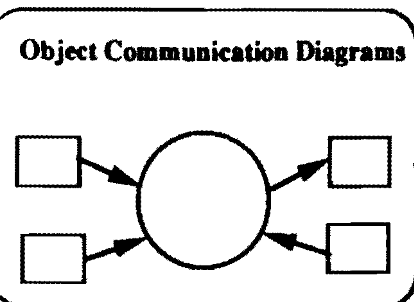
[Parnas 79] Parnas D, "Designing Software for Ease of Extension and Contraction", IEEE Transactions on Software Engineering, March 1979.

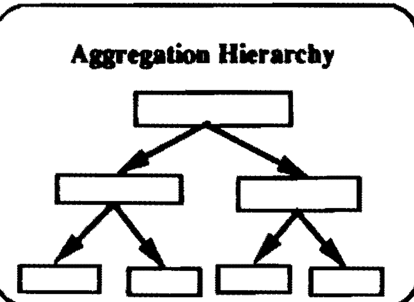
[Pyster 90], Pyster A., "The Synthesis Process for Software Development", in "System and Software Requirements Engineering", Edited by R. Thayer & M. Dorfman, IEEE Computer Society Press, 1990.

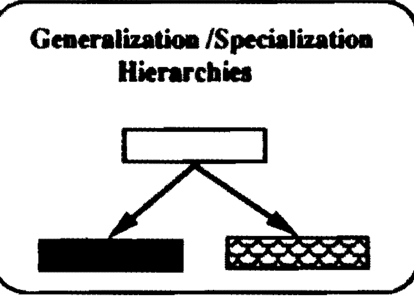
[Szczur 90] Martha R. Szczur, "A user interface development tool for space science systems", Paper presented at the AIAA/NASA Symposium on Space Information Systems, September 1990.

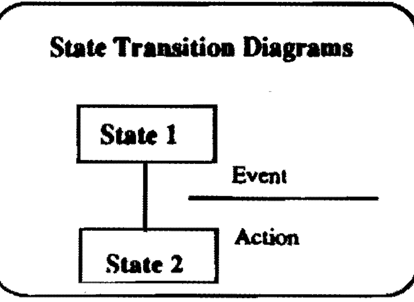
[Yourdon 89] Yourdon E., "Modern Structured Analysis", Prentice Hall, 1989.

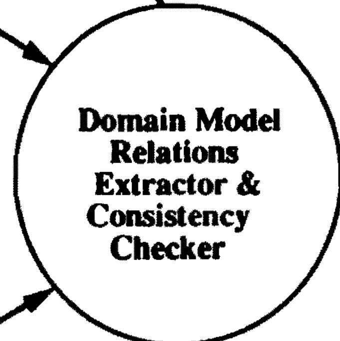
Multiple Views of Domain Model



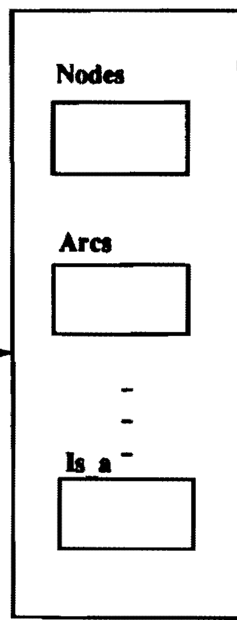


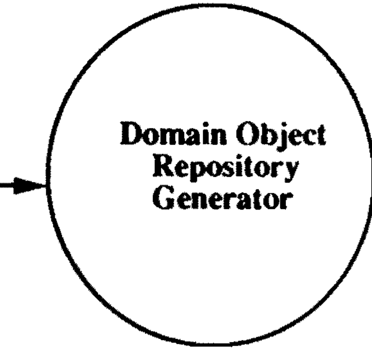






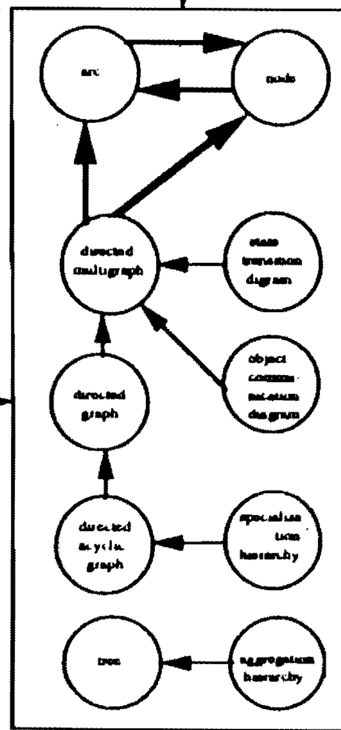
Domain Model Relations

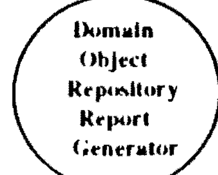




Feature/Object Dependencies

Domain Object Repository





Domain Object Repository Report

Figure 1. Developing the Domain Model

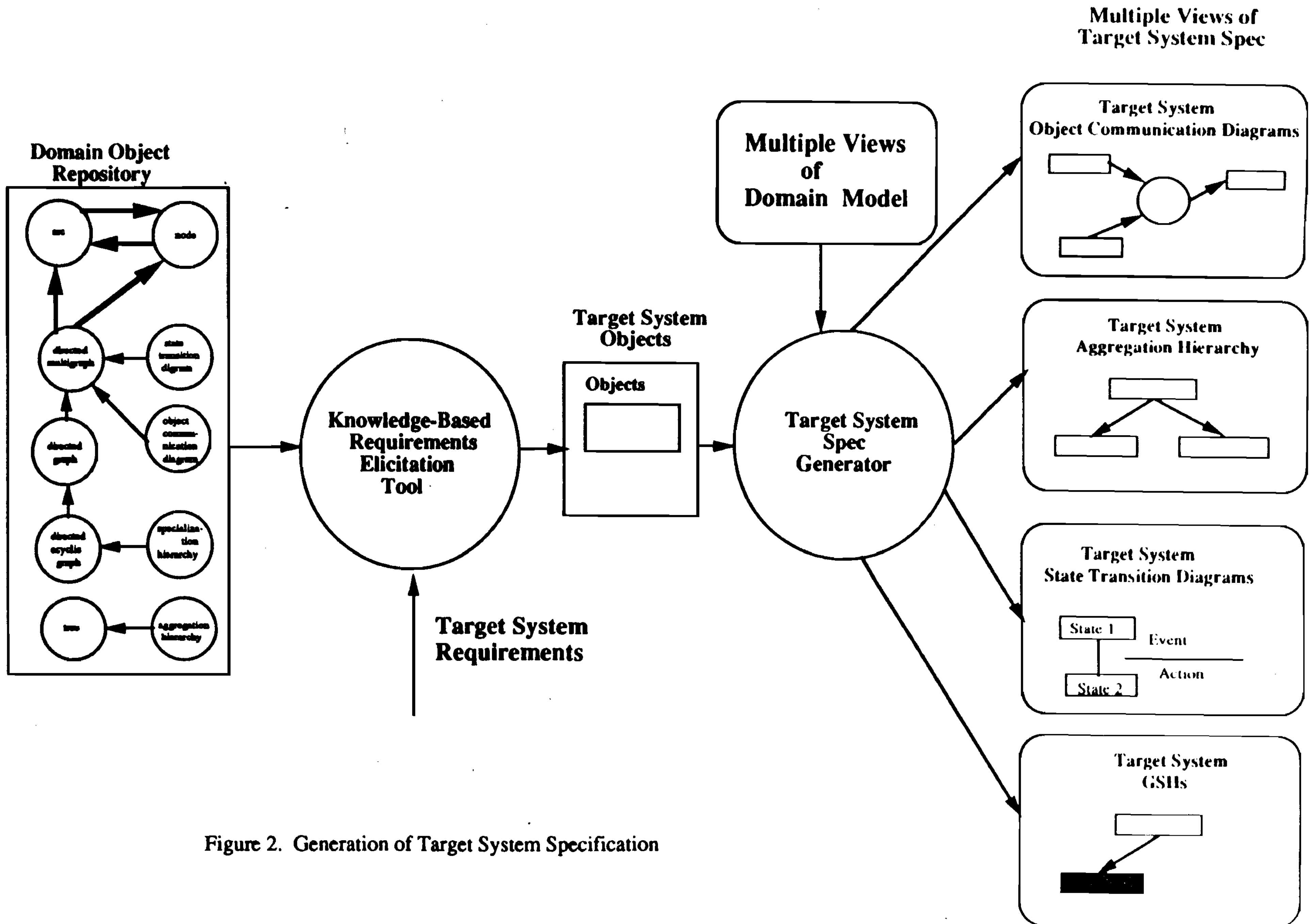


Figure 2. Generation of Target System Specification

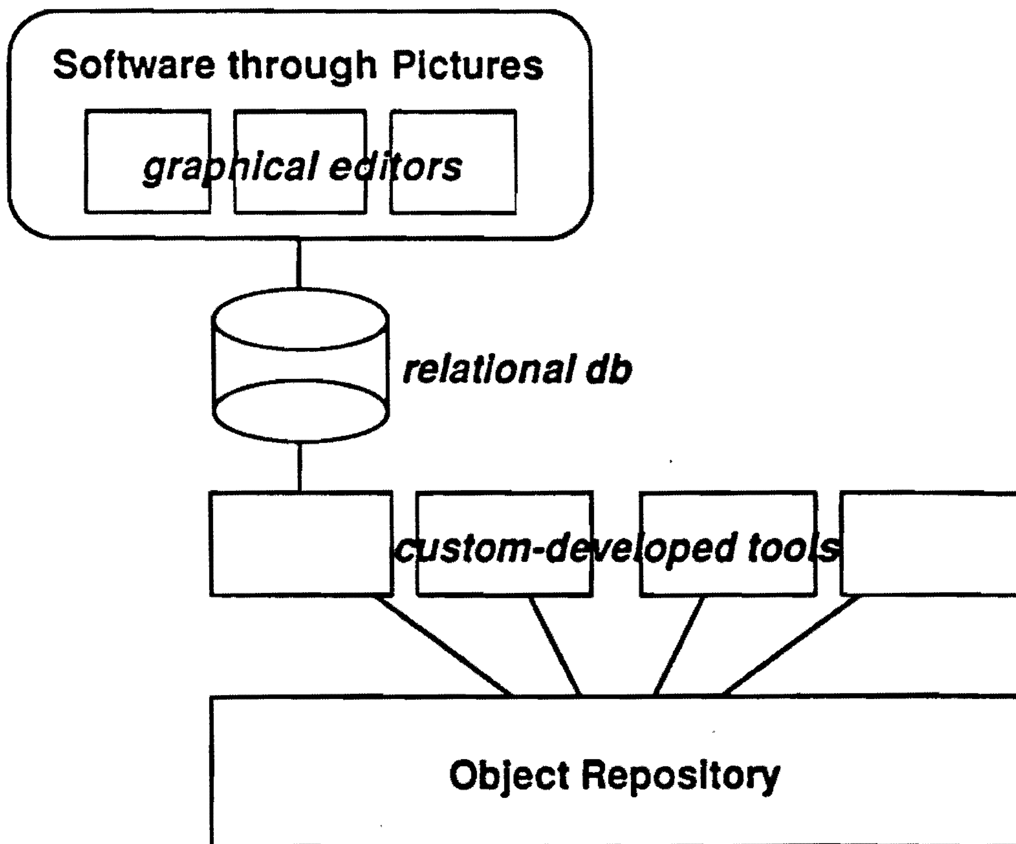


Figure 3: Position of the object repository within overall architecture

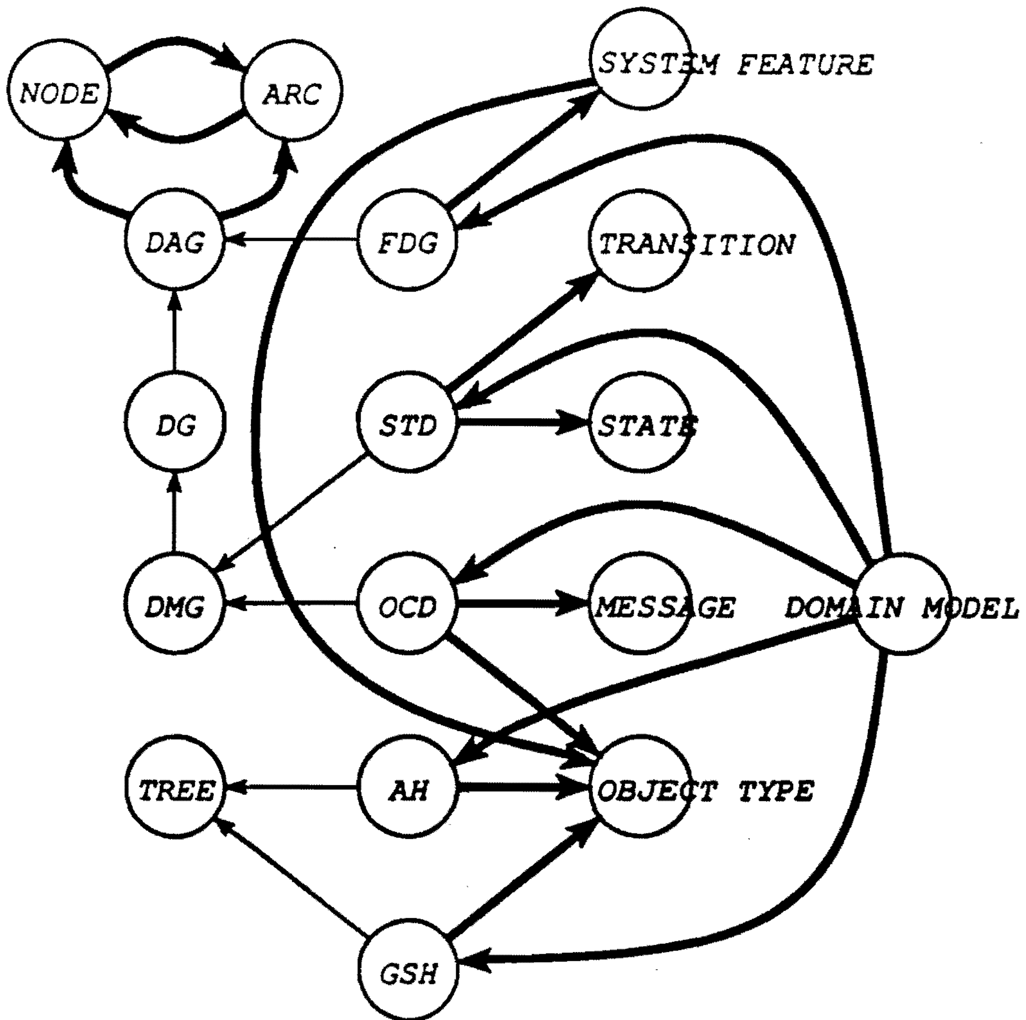


Figure 4: Structural relationships among classes forming the object repository's schema

The DIAGRAMS relation.

diagram name

The NODES relation.

node name	diagram name	index	characteristic	cardinality

The ARCS relation.

arc label	source node name	sink node name	diagram name

Figure 5: Some relations exported from StP

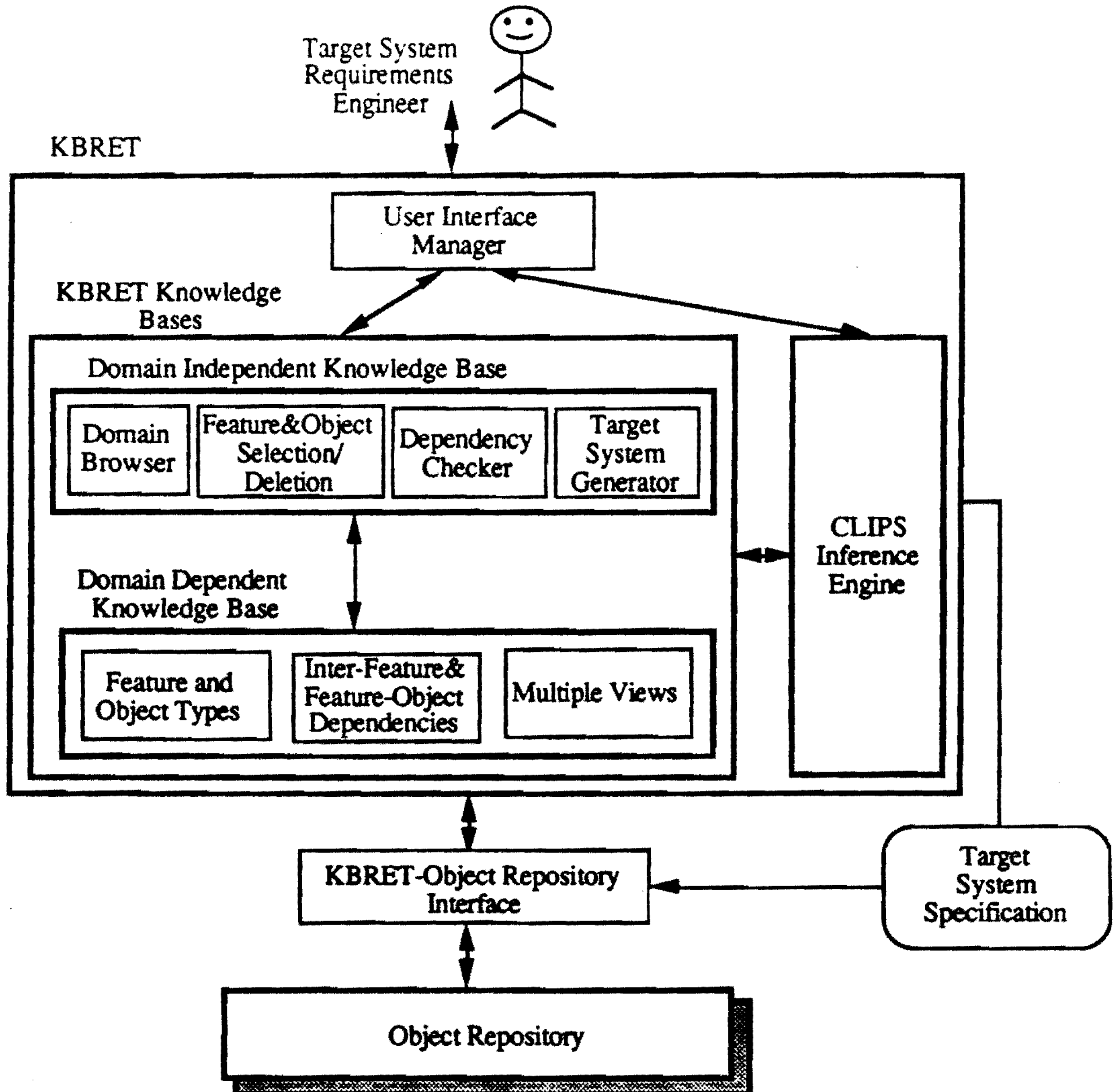


Figure 6. Knowledge Based Requirements Elicitation Tool (KBRET)