IRIS – A Mapping Assistant for Generating Designs from Requirements [1]

Yannis Vassiliou, Manolis Marakakis, Panagiotis Katalagarianos
Lawrence Chung[2], Michalis Mertikas, and John Mylopoulos [2]

Institute of Computer Science
Foundation for Research and Technology - Hellas
Heraklion - Crete
Greece

Abstract

The problem of generating information system designs from requirements specifications is addressed, with the presentation of a framework for representing requirements and a mapping assistant, IRIS[3], that facilitates the design generation process. Requirements are viewed as knowledge bases and the knowledge representation formalism for the prototype, also the language for implementing IRIS, is Telos which provides facilities for describing entities and relationships and for representing and reasoning with temporal knowledge. The generation of a design is achieved with a mapping process from requirements which is: (i) Locally guided by dependency types determining allowable mappings of an element of a requirements model, (ii) globally guided by non-functional requirements, such as accuracy and security requirements on the intended system, represented as goals describing desirable properties of the intended system and used to guide local decisions.

The paper details a prototype implementation (IRIS) of the proposed mapping framework and illustrates its features through a sample session.

[1]This is a report on results from the DAIDA project, funded in part by the European Commission through the Esprit programme under contract no. 892 [Jarke86]; financial support for this research was also received from the Institute of Computer Science of the Foundation for Research and Technology - Hellas (FORTH), the National Science and Engineering Research Council of Canada and the University of Toronto.

[2]Department of Computer Science University of Toronto, Canada

[3]IRIS was a famous Greek goddess, considered to be the personal assistant of Zeus.

1 Introduction

A *requirements model* includes both *functional* and *non-functional* requirements [Roman85]. Functional requirements provide constraints on the functionality of the intended system and for the case of information systems may include (a) a description of the environment within which the intended information system will eventually function, hereafter the *environment model*, (b) a description of the functions carried out by the information system, hereafter the *system model* and (c) a description of the interactions between the intended system and its environment, hereafter the *interaction model* [Borgida89]. Since the subject matter of requirements modelling is some part of the world, it is reasonable to view requirements models as *knowledge bases* which capture knowledge about an environment, e.g., a corporation or an office, but also describe how the intended information system is to be embedded in and interact with that environment ([Zave81] [Jackson83] [Mylopoulos86] also adopt this point of view).

In addition to functional requirements, a requirements model also includes *non-functional requirements* which impose global constraints on the operation, performance, accuracy and security of any proposed solution to the functional requirements model. For example, considering a hypothetical expense report system for research projects, non-functional requirements may require that the intended system run on a PC, and that the expense information be accurate and secure in the sense that it is only available to key persons within each project.

An *information system design* describes the structure of the information managed by the intended system as well as the behavior of the processes manipulating that information. As such, it can be viewed as a formal specification of the system to be built in the spirit of formal specification work [Hayes87]. However, unlike formal specifications intended for other programming tasks, such as the development of an operating system, those of interest here include descriptions of highly complex data structures and generally simple algorithms. *Semantic Data Models* have been offered as extensions of conventional data models appropriate for the development of information system designs [Borgida85]. Such models attempt to capture a human's conceptualization of the structure and behavior of an information system while omitting implementation details.

The generation of a system design involves many refinements in mapping each of the various components of requirements models down to different constituents of system designs. Without adequate guidance on how to make refinements, the generation of a system design is an extremely difficult task. Some of the problems that need to be faced in generating system designs include:

1. *Coping with omissions in functional requirements*: In requirements models, details are omitted concerning entities, cause-effect chains, and requirements violations. Since system designs result from successive refinements of requirements models, we need to discover rules that assist the introduction of the omitted details for various components of requirements models.

2. *Supporting non-functional requirements*: Little work exists on how to use non-functional requirements in the generation of a system design.

3. *Exploiting representational commonalities*: The languages or formalisms chosen for

requirements modelling and system design hopefully may share knowledge representation features. Methods need to be developed for the exploitation of commonalities in order to simplify the mapping process.

4. *Correctness of mapping*: Each design should be consistent to the requirements specification from which it was generated. However, no formal framework is available to date that guards against invalid mappings. Thus, we need to devise ways for ensuring the correctness of mappings.

It should be noted that not all the parts of a requirements model are mapped down to designs. The components of the systems model are only mapped to designs. The rest of the requirements model prescribes the nature of interaction of the information system and its environemnt and the meaning of the information maintained by the system.

The paper presents IRIS, a mapping assistant prototype/demonstrator of a *dependency-based, goal-oriented* methodology to the mapping problem. The methodology is *dependency-based* in the sense that the mapping of parts of the requirements model into a design is guided by predefined allowable dependencies. Data entities in the design, for example, may only be derived from, and therefore depend on, entities in the functional requirements model, while activities may be mapped onto transactions or scripts.

At the same time, the methodology is *goal-oriented* in the sense that non-functional requirements are treated as possibly conflicting goals to be satisfied, to a greater or lesser extent, by the generated design. For each requirement goal, our proposed methodology offers a set of refinement methods to designers to help them guide the mapping process. Each refinement method allows the decomposition of a posted goal into sub-goals and is based on an explicit model for each class of non-functional requirements handled by the methodology (e.g., accuracy, security, operational, performance).

The work reported in this paper was carried out in part within the framework of the DAIDA project, whose goal is to build a software engineering environment for developing and maintaining information systems. Key features of the project are: (a) requirements models and designs are viewed as knowledge bases and representation languages are chosen accordingly, (b) a knowledge base management system which maintains a complete design record for an information system is developed, and (c) mapping methodologies from requirements models to designs to implementations are created.

Telos, TDL (Taxis Design Language) and DBPL[4] are the languages adopted for requirements modelling, design and implementation respectively. The Global Knowledge Base Management System, or GKBMS, manages general knowledge used to guide the users of the environment in the development of a requirements model and in the mapping of that model to a design and later on an implementation. The GKBMS also maintains a history record of decisions and dependencies among requirements, design and implementation components. A detailed description of the DAIDA architecture and the initial aspirations of the project are beyond the scope of this paper and are described in [Borgida89], while the GKBMS component is presented in [Jarke89].

[4]The former languages are introduced in the paper mainly with examples and small explanations. DBPL is a database programming language developed at the University of Frankfurt, which offers a Modula 2-like programming framework extended it with sophisticated relational database management facilities [Schmidt88].

The language for requirements modelling, used in this paper, is *Telos*[Koubarakis89].[5] Telos adopts a representational framework which includes structuring mechanisms analogous to those offered by semantic networks [Findler79] and semantic data models [Hull87]. In addition, Telos offers an assertional sublanguage which can be used to express both deductive rules and constraints with respect to a given knowledge base. Two novel aspects of Telos are its treatment of attributes, promoted to a first class citizenship status, and the provision of special representational and inferential facilities for temporal knowledge. Descriptions in a Telos knowledge base are partitioned into *tokens* and *classes*, depending on whether they represent particular entities, say the person John or the number 23, or abstract concepts, say those of Person or Number. Classes are themselves instances of other more generic classes, namely, *metaclasses* that are, in turn, organized along an instantiation hierarchy.

Apart from this representational framework, shared to a large extend by the adopted design language and many other semantic data models, Telos views a requirements model as an account of a history of events and activities, thus emphasizing the use of *time* in the description of a corporation or office within which the intended information system will function. The model of time adopted is based on time intervals [Allen81]. Every Telos proposition includes, along with other structural information, a temporal interval which specifies the lifetime of the represented entity or relationship. Time intervals are related to each other through temporal relations such as *before, during* and *overlaps* (thirteen possibilities in all) during the description of individuals or through assertions. Inferences with respect to temporal relations are handled by a special inference procedure rather than a general purpose inference mechanism, with obvious performance advantages. In addition, each Telos expression has a temporal component which acts as a filter on its possible values.

Design specifications, according to the DAIDA world view, present a conceptual view of the information system by structuring the data and transactions which constitute the system according to their intended meaning rather than their implementation. TDL (short for Taxis Design Language) facilitates the development of such specifications by offering a uniform semantic data model for describing data, transactions and long-term processes [Borgida89]. As with many other semantic data models, the one adopted here offers the notions of entity and relationship along with aggregation, generalization and classification intended as structuring mechanisms.

TDL offers a variety of *data classes* for modelling the entities that are relevant to the application domain and at the same time will eventually be stored in the database(s) of the information system. Data classes include as special cases conventional data types (Integer, String, enumerated and subrange types), but also labelled Cartesian products *aggregate classes*, whose instances have equality decided structurally, and *entity classes* which have their extensions (collections of instances) externally updated.

Each type of data class has associated *attribute categories* which indicate the kinds of attributes that are applicable to their members. It should come as no surprise that subclass hierarchies are supported and their use is encouraged throughout the design. Note that as a specification language, TDL makes no commitment on how such attributes are to be stored (arrays, records, relations), nor whether the information provided by the

[5]Telos has evolved from CML ([Stanley86]) which, in turn, is an enhanced version of RML, a requirements modelling language proposed in ([Greenspan84]).

attribute will be obtained by look-up or computed by a function. It is the prime goal of the design-to-implementation mapping process to make and justify these decisions.

Turning to its procedural sublanguage, TDL offers functions and two types of procedures that effect state changes: *transactions*, intended to define atomic database operations, and *scripts*, intended to model long term processes. Each transaction is supposed to specify a set of allowable state transitions through pre/postcondition constraints on the values of the state variables (attribute functions, parameters and class extensions).

TDL assumes frame axioms, i.e., state components that are not affected by a transaction, are assumed to remain unchanged. This is one of the fundamental differences between TDL and Telos.

As indicated earlier, to model activities with prolonged duration (e.g., running a project) as well as to describe the system's interaction with its users, TDL supports the notion of *scripts* [Barron82], [Chung84]. A script is built around a Petri-net skeleton of states connected by transition arcs which are augmented by condition-action rule pairs. The rules allow reference to the passage of time, and permit the exchange of messages following Hoare's CSP mechanism.

While it is beyond the scope of this paper to fully and formally introduce Telos and TDL, it unavoidably dwells into some notational details of the two languages while describing the mapping assistant. Effort is made to keep this to a minimum without losing scientific validity and content.

Section 2 describes the dependency-based, goal-oriented mapping framework adopted and analyses some of the technical problems that arise in mapping requirements to designs. Section 3 goes over an example session using IRIS to give a feel of the framework and the facilities that could be expected. A functional requirements model, concerning project expense accounts, which is used as running example throughout the paper is also presented. Major mapping issues such as the problem of representation are discussed in Section 4. Finally, Section 5 presents the status of the implementation and some concluding remarks.

2 IRIS: A Prototype Mapping Assistant

This paper emphasizes the dependency-based aspect of the mapping problem, while more details on the complete goal-oriented methodology and the use of non-functional requirements for the mapping process can be found in [Chung89].

The role of IRIS is to assist the designer who makes the final decisions. Dependency types define the options available to him/her, while dependencies relate design objects and the corresponding requirements objects. Automatic selection of dependencies is performed only in situations where there is a single applicable dependency and the system need only inform the designer of the selected dependency. The mapping assistant interface employs a graphical presentation of objects and dependencies at two different levels of detail.

2.1 System Architecture

The mapping activity transforms the system model component of a requirements specification into a conceptual design. A fundamental consideration throughout the mapping process is the interplay of requirements entities and activities with corresponding design data

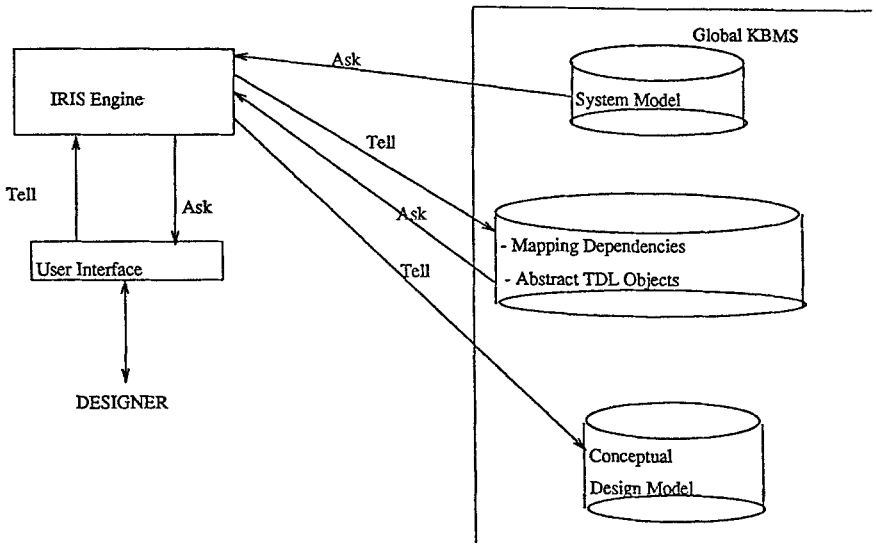

Figure 2.1 General Architecture of the Mapping Assistant

classes, transactions and scripts. The dependencies and dependency types used by the mapping assistant are represented in Telos, along with requirements and design objects. IRIS, the mapping assistant, assumes the following types of dependencies: classification (dealing with mappings of classes), attribute (dealing with the mapping of attributes), IsA (for IsA hierarchies), and instantiation (for instantiation hierarchies). Figure 2.1 depicts the general architecture of the mapping assistant, the knowledge relevant to the mapping task and the assistant's interaction with the user.

2.2 Representation of Design Objects in Telos

The features, syntax and semantics, of the conceptual design language TDL have been modeled as Telos metaclasses organized along isA hierarchies. These metaclasses have as instances classes which represent components of the conceptual design, i.e., TDL classes. All these individual and attribute metaclasses are instances to the `omega_class`, `TDL_Object` and will be referred to as *abstract TDL objects.* At the top of the metaclass isA hierarchy is `TDL_MetaClass` with specialized subclasses `TDL_DataClass`, `TDL_Procedure` and `TDL_Script`.

Generally, the Telos representation of TDL designs encompasses two layers, the first consisting of metaclasses modelling features of the design language (TDL) and is application-independent, while the second consists of simple classes representing the conceptual design model for a particylar application. The organization of abstract TDL objects is depicted in Figure 2.2.

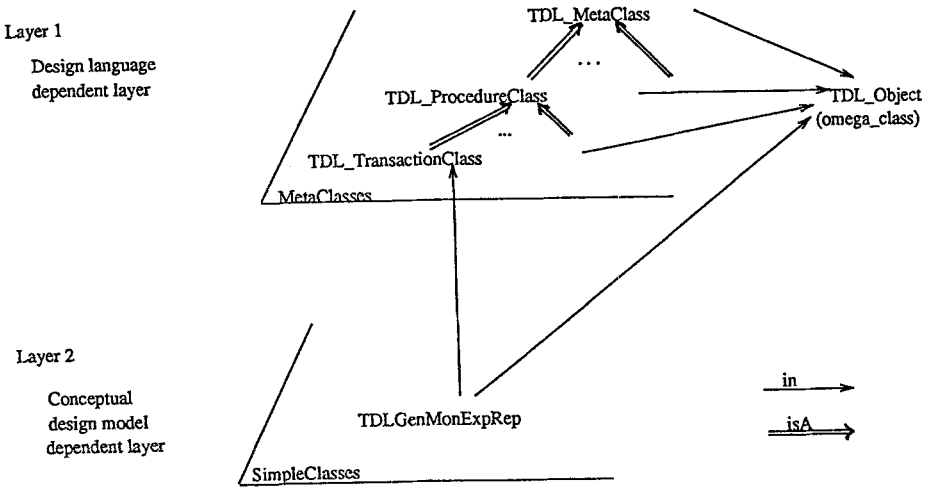

Figure 2.2 Two-layer representation of the design model in Telos

The discussion that follows focuses on the modelling of attribute categories of the design language. These have been defined as Telos attribute metaclasses organized along isA hierarchies. `TDL_AttributeClass` is treated here as the most general attribute metaclass with specializations: `TDL_DataAttributeClass`, `TDL_TransactionAttributeClass` and `TDL_ScriptAttributeClass`. These metaclasses are further specialized to lead to definitions of TDL attribute categories.

```
AttributeClass TDL_AttributeClass in M1_Class, TDL_Object with
    components
        from: TDL_MetaClass;
        label: String;
        to: TDL_MetaClass;
        when: AllTime;
end TDL_AttributeClass
```

The component attributes of this definition indicate the internal structure of the proposition `TDL_AttributeClass` [6] The Telos attribute class `Produces` shown below is the attribute metaclass modelling the TDL attribute category `Produces`. The `from` component of each attribute class specifies the valid objects where that attribute class can

[6]The internal structure of every Telos proposition includes four components, labelled respectively *from, label, to, when.* Since the *when* component of all definitions below is *AllTime*, it will be omitted from definitions that follow.

be used. In particular, Produces can be used as attribute category only in instances of TDL_TransactionClass.

```
AttributeClass TDL_TransactionAttributeClass
                              in M1_Class
                              isA TDL_AttributeClass with
    components
        from: TDL_TransactionClass;
        label: String;
        to: TDL_MetaClass;
end TDL_TransactionAttributeClass

AttributeClass Produces in M1_Class, TDL_Object
                         isA TDL_TransactionAttributeClass with
    components
        from: TDL_TransactionClass;
        label: produces;
        to: TDL_Object;
end Produces
```

The *not setOf* feature of TDL is represented in the model of abstract TDL objects by the built-in Telos attribute category `single`, an attribute which is not single is by default setOf.

Justification is another interesting attribute metaclass which relates each abstract TDL object with an instance of the selected mapping dependency, which effected the creation of that object.

For instance, using some the constructs discussed, the TDL transaction `GenMonExpRep`

```
TRANSACTION GenMonExpRep WITH
   IN
     exp: SETOF Expense;
   LOCALS
     pr : Project;
     m: Month;
     per: String;
   CONSUMES
     exp: SETOF Expense;
   PRODUCES
     expRep: MonthlyEmplExpenseReport;
   GOALS
     : (expRep.mo' = m) AND (expRep.amount' = SUM(exp))
       AND (expRep.proj' = pr)
END
```

is represented in Telos by the class `TDLGenMonExpRep` as demonstrated by:

```
IndividualClass TDLGenMonExpRep
                in S_Class, TDL_TransactionClass, TDL_Object with
    justification
        : MonMbrExpRptToTDLGenMonExpRep_Dep
    tdl_in
        exp: TDLExpenses
    locals, single
        pers: TDL_String;
        m: TDLMon
    produces, single
        exr: TDLMonEmplExpRep
end TDLGenMonExpRep
```

2.3 Representation of Mapping Dependencies

Telos classes and metaclasses model the mapping dependencies between a requirements model and a corresponding conceptual design. The dependency metaclasses, modelling dependency types, define conditions under which there can be a dependency between a requirements and a design object. For the purposes of the prototype system, a dependency name, by convention, indicates the type of both the source requirements object and the target TDL object. For instance, the dependencies `Activity_Transaction_Dep` and `NecessarySingle_Unchanging_Dep` model the mapping of Telos activity classes into TDL transactions and the mapping of a Telos `necessary` and `single` attribute to a TDL `unchanging` attribute respectively. Every dependency metaclass has at least two necessary attributes named respectively `telosObject` and `abstractTDLObject` which indicate the corresponding requirements and design objects.

Dependency metaclasses for Telos activities and TDL transactions are shown below. The attribute `telosObject` in the dependency `Activity_Transaction_Dep` is inherited from `Activity_Procedure_Dep`. The integrity constraint in `Activity_Procedure_Dep` states that the life of the instances of `Activity_Procedure_Dep` co-end with the life of the transformed activity. The mapping of the attributes of an object from the requirements model are represented by the attribute `attributeDep`. All the dependency classes are instances of the omega_class `Omega_Dep`. The most generalized dependency metaclass is `Telos_TDL_Dep`. The attribute `isaDep` models the mapping of isA relationships.

```
IndividualClass Telos_TDL_Dep in M1_Class, Omega_Dep with
    necessary
        telosObject: SystemClass;
        abstractTdlObject: TDL_MetaClass
    attribute
        isaDep: ISA_Dep
end Telos_TDL_Dep

IndividualClass Activity_Procedure_Dep
```

```
                              in M1_Class, Omega_Dep,
                              isA Telos_TDL_Dep with
    necessary
        telosObject: ActivityClass;
        abstractTdlObject: TDL_ProcedureClass
    integrityConstraint
        : $(Forall x/Activity_Procedure_Dep)
           (coends(when(x), when(x.telosObject))$
end Activity_Procedure_Dep

IndividualClass Activity_Transaction_Dep
                              in M1_Class, Omega_Dep
                              isA Activity_Procedure_Dep with
    necessary
        abstractTdlObject: TDL_TransactionClass;
        attributeDep: TransactionAttribute_Dep
    attribute
        isaDep: Activity_Transaction_ISA_Dep
end Activity_Transaction_Dep
```

The dependency OutputSingle_Produces_Dep models the mapping of an output attribute from a Telos activity class to a produces attribute in a TDL transaction. The integrity constraint in OutputSingle_Produces_Dep, states that before mapping an attribute of a Telos object the type of that attribute, an attribute class, should have been mapped. Attribute_Dep is the most general dependency metaclass for attributes.

```
IndividualClass Attribute_Dep in M1_Class, Omega_Dep
    attribute
        telosAttribute: AttributeClass;
        abstractTdlAttribute: TDL_AttributeClass
end Attribute_Dep

IndividualClass  TransactionAttribute_Dep
                               in M1_Class, Omega_Dep
                               isA Attribute_Dep
end TransactionAttribute_Dep

IndividualClass ActivitySingle_Dep in M1_Class, Omega_Dep
                                    isA TransactionAttribute_Dep
end ActivitySingle_Dep

IndividualClass ActivitySetOf_Dep in M1_Class, Omega_Dep
                                   isA TransactionAttribute_Dep
end ActivitySetOf_Dep

IndividualClass OutputSingle_Produces_Dep
                                  in M1_Class, Omega_Dep
```

```
                                isA ActivitySingle_Dep with
    necessary
        telosObject: Output;
        abstractTdlObject: Produces
   integrityConstraint
       :$ (Forall x/OutputSingle_Produces_Dep)
          (Exists existingDep/Activity_Transaction_Dep)
          (existingDep.telosObject=from(x.telosObject) and
          (existingDep.attributeDep = x))$
end OutputSingle_Produces_Dep
```

According to these definitions, the mapping of the activity class &MonMbrExpRpt

```
IndividualClass &MonMbrExpRpt
                          in S_Class, ActivityClass, SystemClass
                          isA &GenExpenseReport with
    input
         exp: &Expense
    control
         pers: &Person;
         m: &Month
    output
         exrep: &MonEmplExpRpt
    activationCondition
         : (Exists t/Date)(now during t and
                           LastDayOfMonth(t, m))
end &MonMbrExpRpt
```

leads to the following dependencies:

```
IndividualClass MonMbrExpRptToTDLGenMonExpRep_Dep
                         in S_Class, Activity_Transaction_Dep
                         isA Dep_S_Class with
    telosObject
        : MonMbrExpRpt
    abstractTdlObject
        : TDLGenMonExpRep
    attributeDep
        : MonMbrExpRpt_exp_Dep;
        : MonMbrExpRpt_pers_Dep
        : MonMbrExpRpt_m_Dep;
        : MonMbrExpRpt_exrep_Dep
end MonMbrExpRptToTDLGenMonExpRep_Dep
```

```
IndividualClass MonMbrExpRpt_exrep_Dep
                        in S_Class, OutputSingle_Produces_Dep
                        isA Dep_S_Class with
    telosObject
        : &MonMbrExpRpt!exrep
    abstractTdlObject
        : TDLGenMonExpRep!exr
end MonMbrExpRpt_exrep_Dep
```

Note that dependency metaclasses, as defined here, are independent of the application domain and only depend on the nature of requirements and design specifications. Selecting a dependency type results in the creation of an instance to the corresponding dependency metaclass.

3 A Sample Session

The prototype implementation of the mapping assistant IRIS was developed with the following requirements in mind:

- Assistance to the designer with the decisions he needs to make during the mapping process.
- Maintenance of the history of the mapping process.
- Offering simple-to-use tools that aid the mapping process.

Figure 3.1 shows the interface of IRIS, consisting of three separate areas:

- The Telos area, which provides access to Telos objects. (More precisely, the SML area, where SML is an extension of Telos with built-in attribute categories, Telos and SML are used interchangeably)
- The TDL area, which provides access to TDL objects.
- The dependency types area, which provides information on available dependencies.

Early on in the implementation of the mapping assistant, and consistently with other components of the DAIDA project, it was decided to use graphical representation to help the designer have a better view of the contents of the requirements and design specifications as well as the mapping process. Moreover, node highlighting and dynamic pop-up menus were adopted as display methods to suggest to the designer possible choices. In addition, the mapping assistant is equipped with a powerful TDL editor which minimizes the amount of information to be provided by the designer during the mapping of a requirements object into a design one.

Consider the following example, for the purposes of demonstrating the mapping process. Suppose that the designer decides to map the following isA hierarchy which is part of the requirements model:

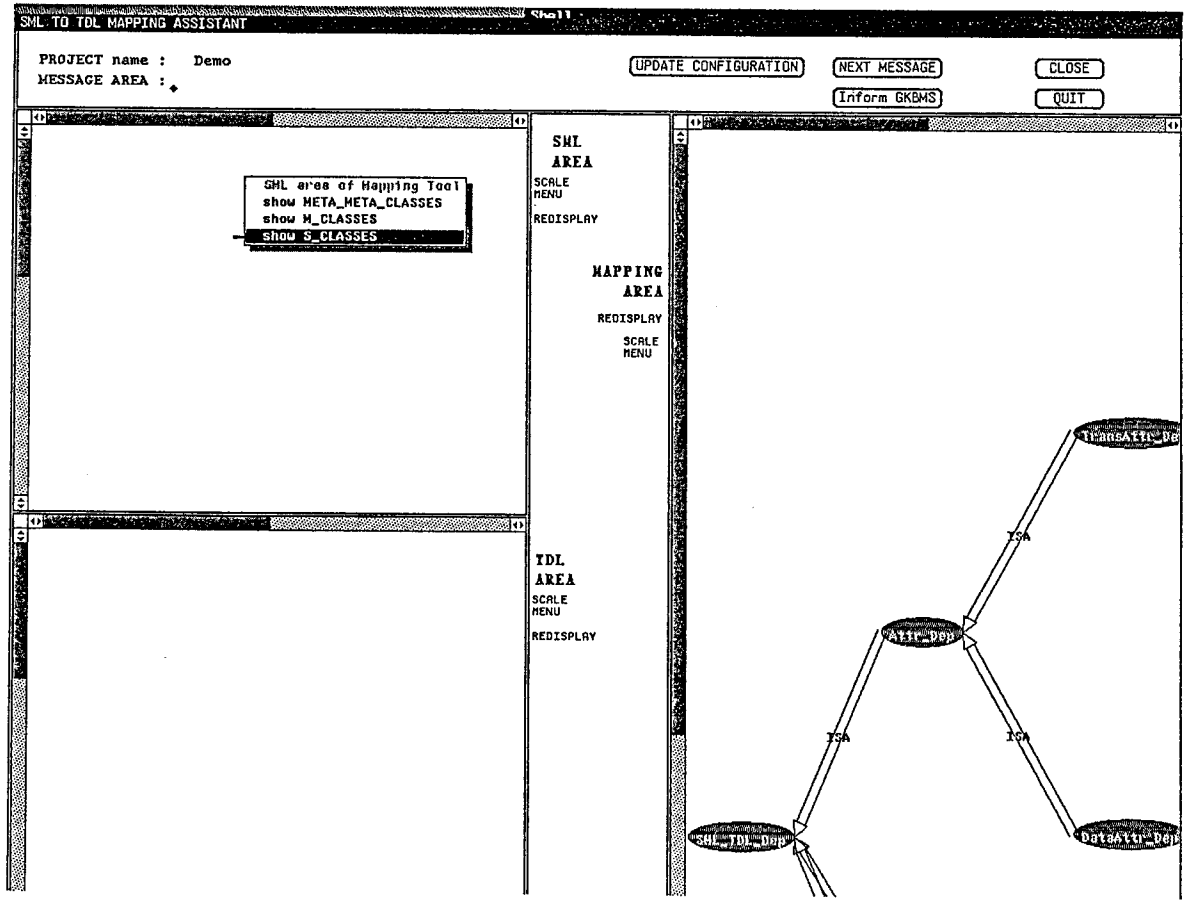

Figure 3.1 The Mapping Assistant Interface

```
IndividualClass &GenExpenseReport
                in S_Class, ActivityClass, SystemClass with
    input,single
        proj: &Project

            ...

end &GenExpenseReport


IndividualClass &MonMbrExpRpt
                  in S_Class, ActivityClass, SystemClass
                  isA &GenExpenseReport with
    input
        exp: &Expense
    control,single
        m: &Month;
        pers: &Person

            ...

end &MonMbrExpRpt
```

This hierarchy consists of two activities. The most general activity `&GenExpenseReport` generates project reports and a special case of this is the generation of monthly expense reports for employees expressed in our example by activity `&MonMbrExpRpt`.

There are two different ways to map this hierarchy.

One way is to use the isA hierarchy provided by TDL. In this case we simply map each Telos Activity to a TDL Transaction as follows:

```
TRANSACTION GenRep WITH
    IN
        pr: Proj;

          ...

END


TRANSACTION GenMonExpRep ISA GenRep WITH
    IN
        exp: SETOF Expense;
    LOCALS
        m: Mon;
        per: String;

            ...
```

```
END
```

An alternative way is to decide to eliminate this hierarchy from the design specification and to create instead the following transaction:

```
TRANSACTION GenMonExpRep WITH
    IN
        pr: Proj;
        exp: SETOF Expense;
    LOCALS
        m: Mon;
        per: String;

            ...

END
```

This alternative combines mapping of a requirements object with the projection of attributes to all specializations of a class and is referred to as *mapping with inheritance* in [Katalagarianos89]. For the example, `&MonMbrExpRpt` is mapped, with its own and inherited attributes, while `&GenExpenseReport` is not. The particular steps required for this mapping task are as follows:

1. *Selection of the level of the objects to be mapped.*

Each mapping task begins with the selection of a requirements object. Preliminary to this step, a pop up menu allows the designer to select the classification level of the object to be mapped by choosing

- Simple Classes.
- M1 Classes.
- Meta Meta Classes which consist from M2 classes up to Omega classes.

Action: Select Simple Classes item, figure 3.1.

Effect: All simple classes are presented graphically in the Telos area.

2. *Selection of the object to be mapped*

Next, a specific object is selected for mapping. To complete the mapping, the designer needs to first review its complete description. By clicking the right button of the mouse on a node which represents a particular class, a pop-up menu appears on the screen. Figure 3.2 shows the menu corresponding to class `&MonMbrExpRpt`. As `&MonMbrExpRpt` is a specialization of `&GenExpenseReport`, two different alternatives are offered to the designer for the mapping. Either he can choose the item *start mapping/show dep* which corresponds to the case of mapping the Telos isA hierarchy into a corresponding TDL one, or he can choose the item *mapping with inheritance/show dep.* Both items contain *show dep*, because if the selected class (or hierarchy) has already been mapped then the dependency graph corresponding to that previous mapping is going to be displayed.

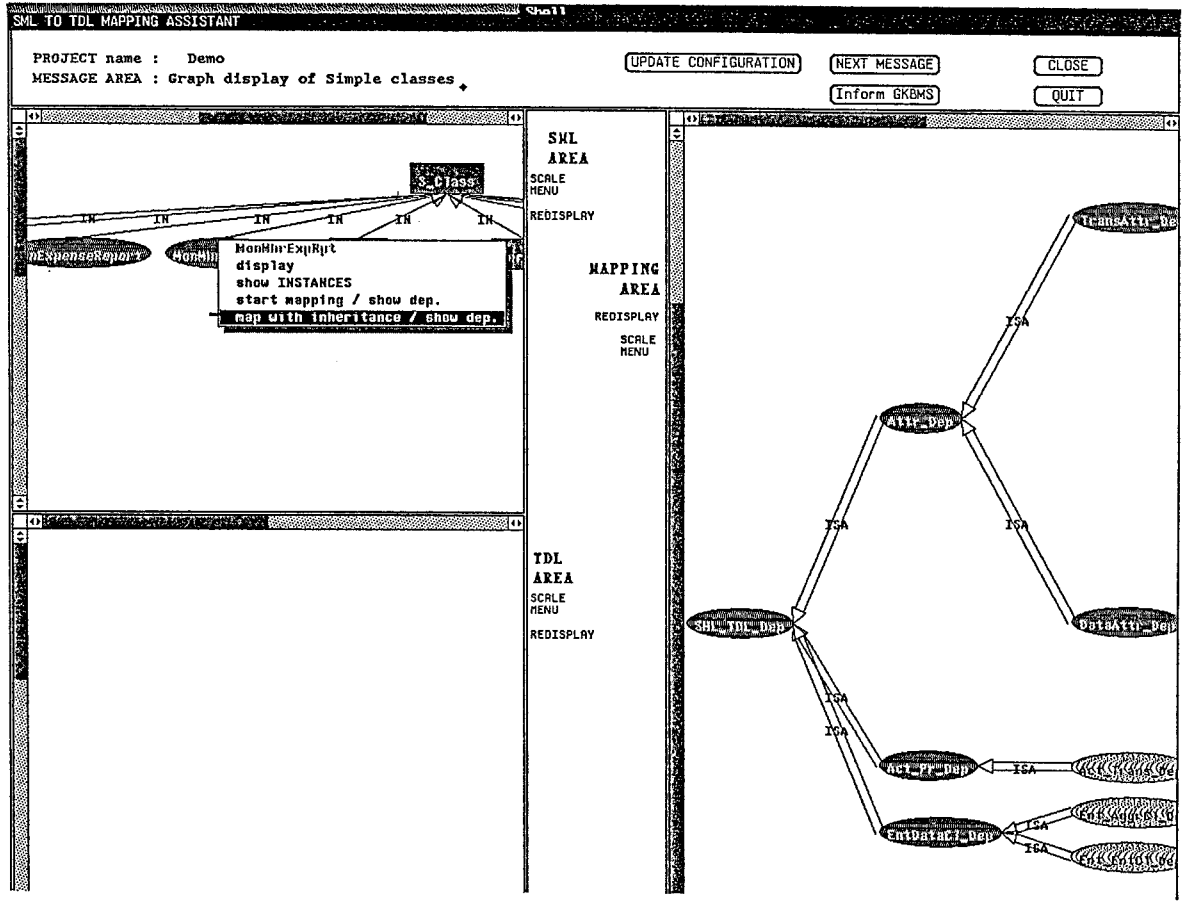

Figure 3.2 Telos Object Selection

Action: Select *mapping with inheritance*

Effect: Attribute displayer comes up (Figure 3.3). The inherited attributes are colored grey.

In order to map a class, the types of its attributes must have been mapped before. If this is not the case, the mapping assistant suspends the mapping process. The special sign "M" on some nodes on the attribute displayer shows that the classes represented by these nodes have been mapped.

3. *Start the mapping process for the object*

The next step is to decide how that object is going to be mapped without having in mind whether it has attributes or not for the time being. By clicking the right button of the mouse on the class node `&MonMbrExpRpt` on the attribute displayer figure 3.3, a pop-up menu comes up with the item *map*.

Action: Select *map*

Effect: Some nodes of the mapping rules area are turned into grey (highlighted), figure 3.4.

4. *Selection of the mapping rule to be used*

The designer has to decide which rule to chose from the grey ones figure 3.4. The system has highlighted the applicable ones for this mapping task. By clicking the appropriate highlighted node, a pop_up menu comes up.

Action: Select *Fire*

Effect: A TDL editor comes up with the frame of the TDL object to be created, figure 3.5.

5. *Start creating TDL Object*

The designer just fills up the slot with the name of the new TDL object.

Action: Click *create abstr* button of the TDL editor.

Effect: The dependency corresponding to the mapping performed is shown on the attribute displayer, figure 3.6.

6. *Continue with attribute mapping*

The next step is to proceed with the mapping of the attributes including the inherited ones.

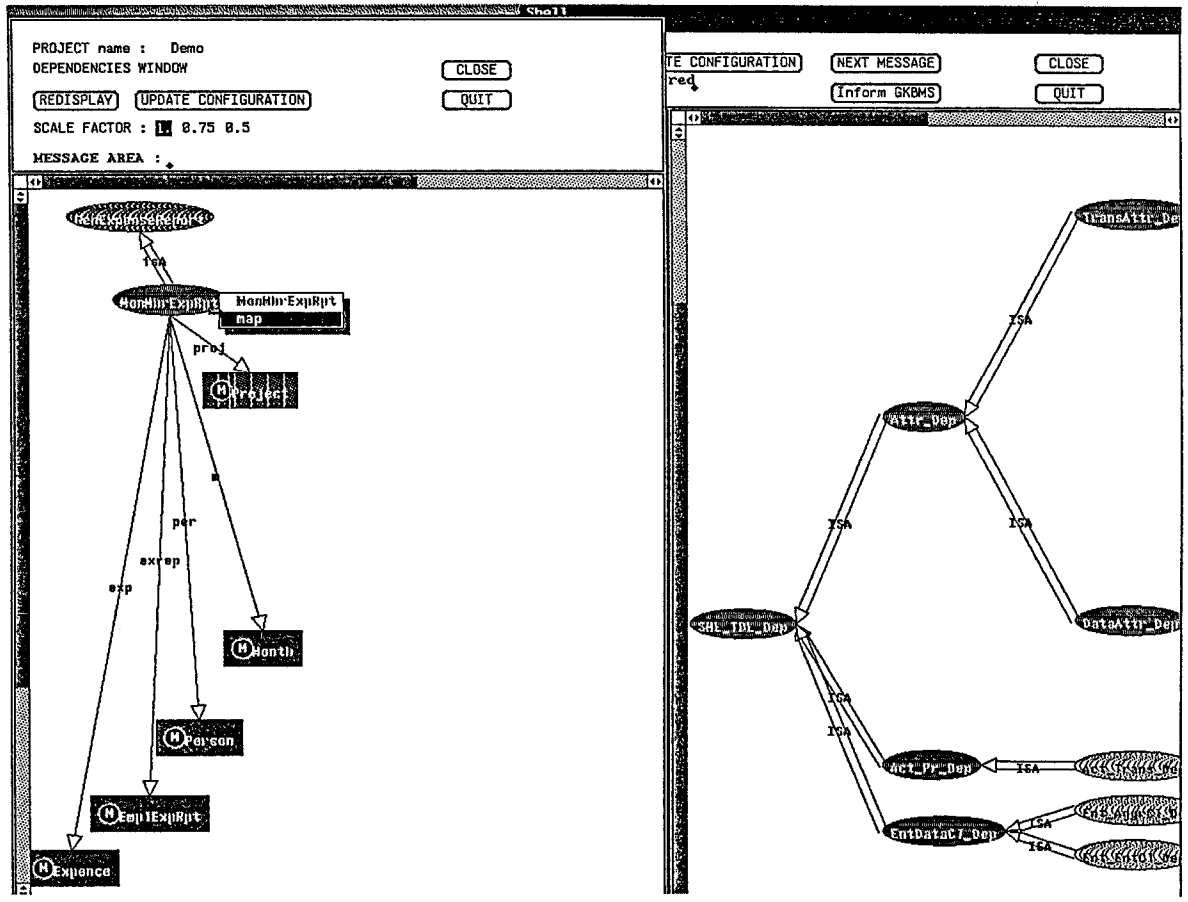

Figure 3.3 Attribute Displayer

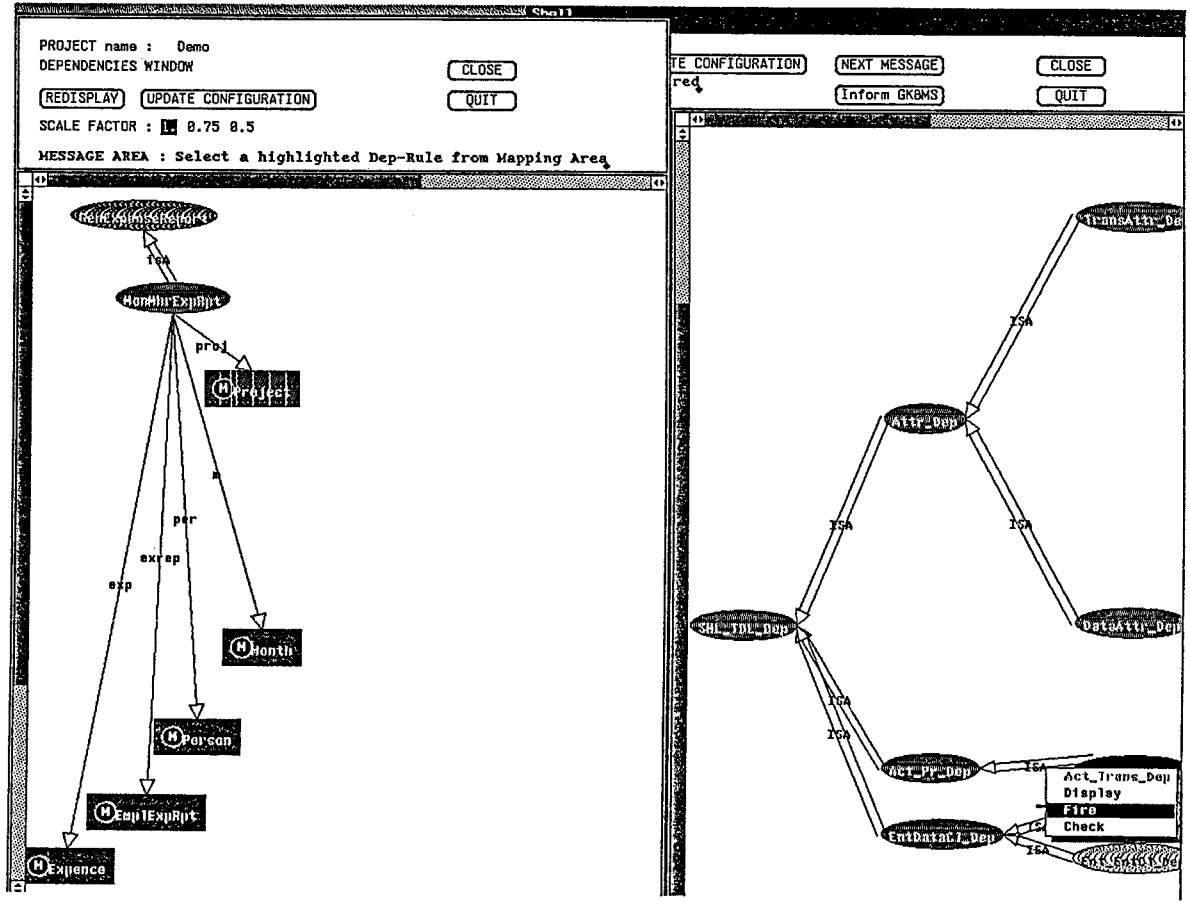

Figure 3.4 Dependency Hierarchies and Dependency Selection

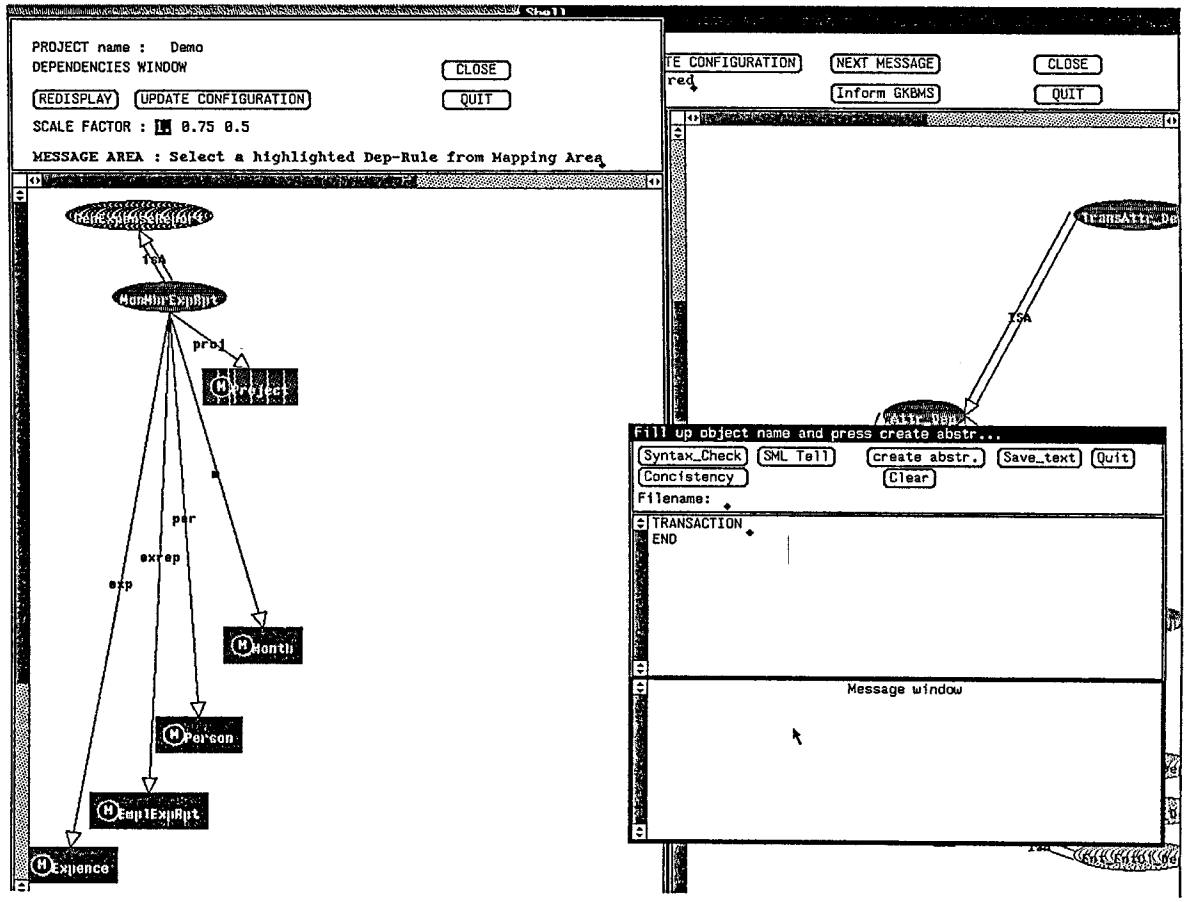

Figure 3.5 TDL Editor

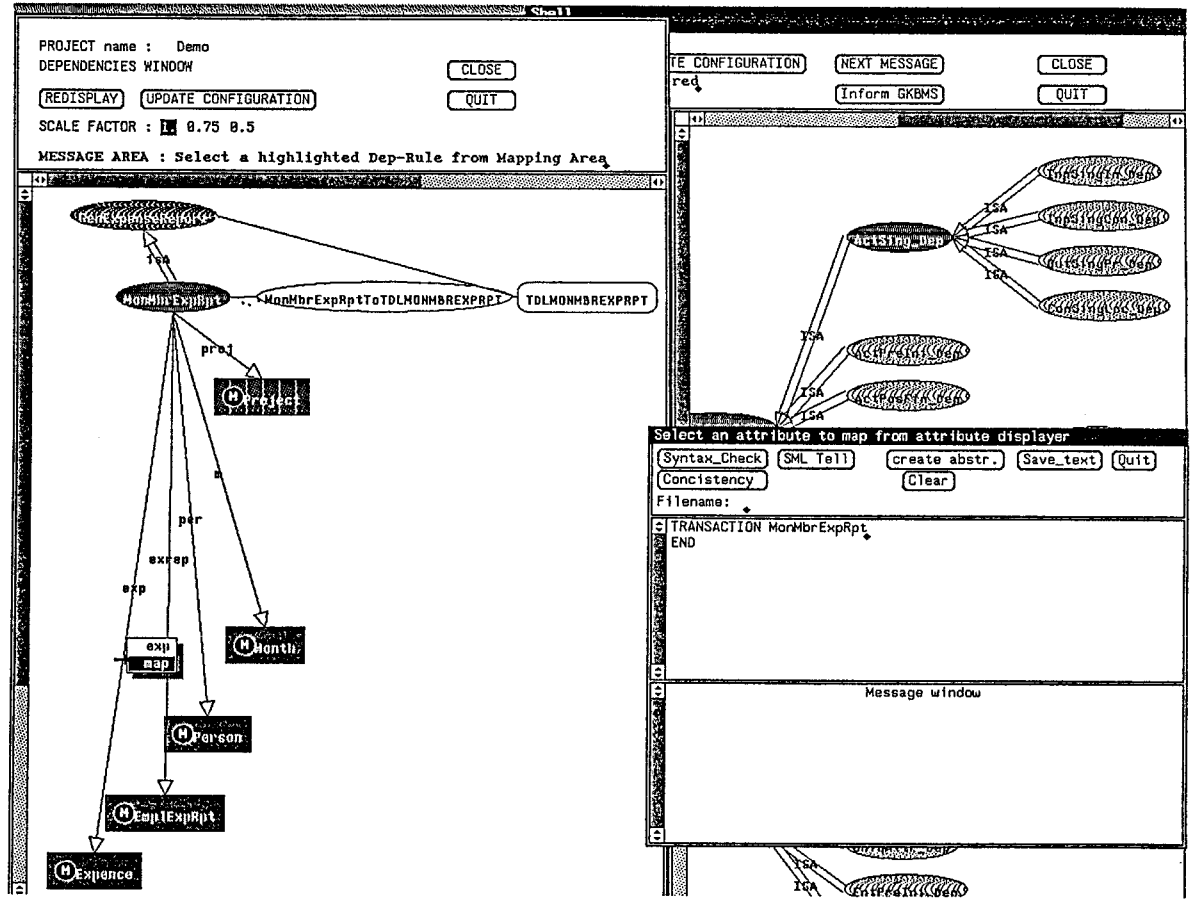

Figure 3.6 Dependency Graph

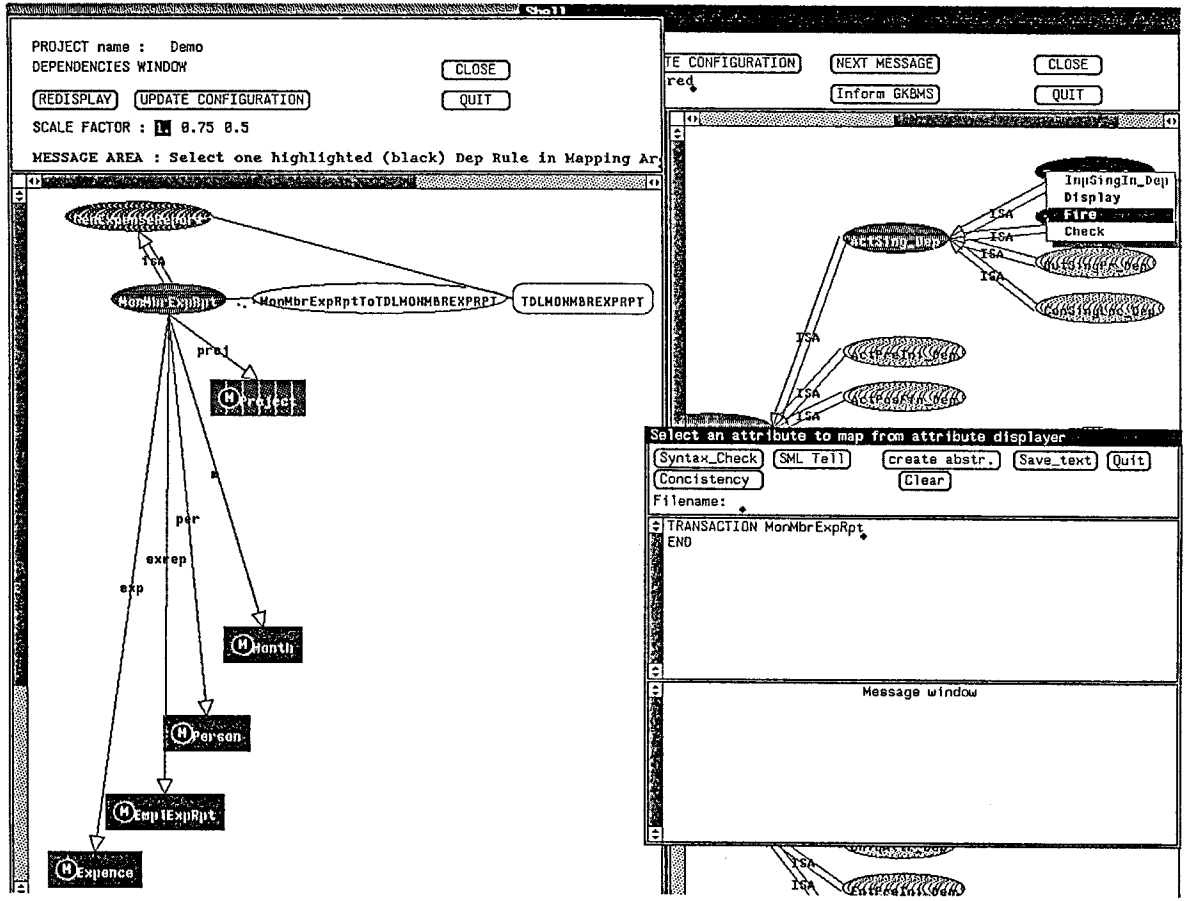

Figure 3.7 Dependencies for Attributes

Action: Click on an attribute link and select the only item *map* of the pop-up menu.

Effect: Some nodes in the mapping rule area are highlighted to grey, figure 3.7.

7. *Selection of the mapping rule to be used for attribute mapping*

Action: Select *Fire* by clicking the desirable highlighted node.

Effect: The form of the TDL attribute appears on the TDL editor.

The TDL attribute type is provided by the mapping assistant. The designer only gives the attribute name.

Action: Fill up the attribute name and press *create abstr.*

Effect: The dependency corresponding to the mapping of attribute is shown in the attribute displayer, figure 3.8.

Action: Repeat steps 6) and 7) until all the attributes have been mapped.

After all attributes are mapped the abstract TDL object and dependency instances corresponding to this mapping have been created.

Action: Press the button *SML Tell* of the TDL editor.

Effect: Abstract TDL object and dependency instances are being *Told* to TELOS KB.

Action: Press *Syntax Check* and then *Consistency* on the TDL editor.

Effect: TDL object is being *Told* to TDL KB. Class `&MonMbrExpRpt` is marked as mapped on the attribute displayer, and mappping is over. Figure 3.9 shows the corresponding dependency graph.

Special handling is offered by the mapping assistant when a requirements object that has already been mapped needs to be changed. In order to maintain consistency, all Telos objects influenced by this change have to be remapped. The designer does not have to follow the mapping steps described previously in order to remap these objects. Instead the mapping assistant finds the objects that have been affected by the change and asks the designer to take specific actions. For example, it may be recommended that the designer remove the changed objects and remap the new ones, or change specific objects created by the old mapping process without having to remap the changed object. The actions to be taken by the designer in such circumstances are intended to preserve consistency, of the requirements specification, the design specification or the mapping process.

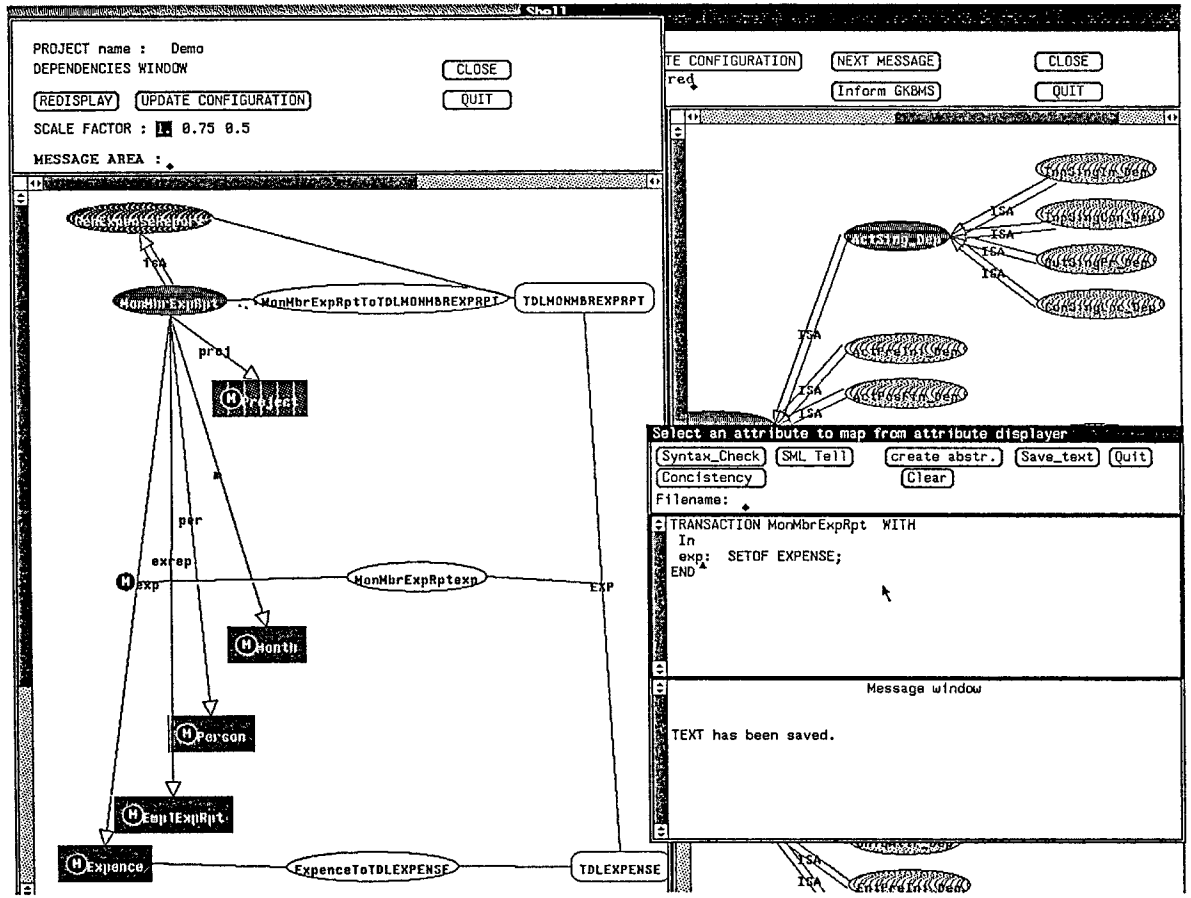

Figure 3.8 Dependency Graph Including Mapped Attributes

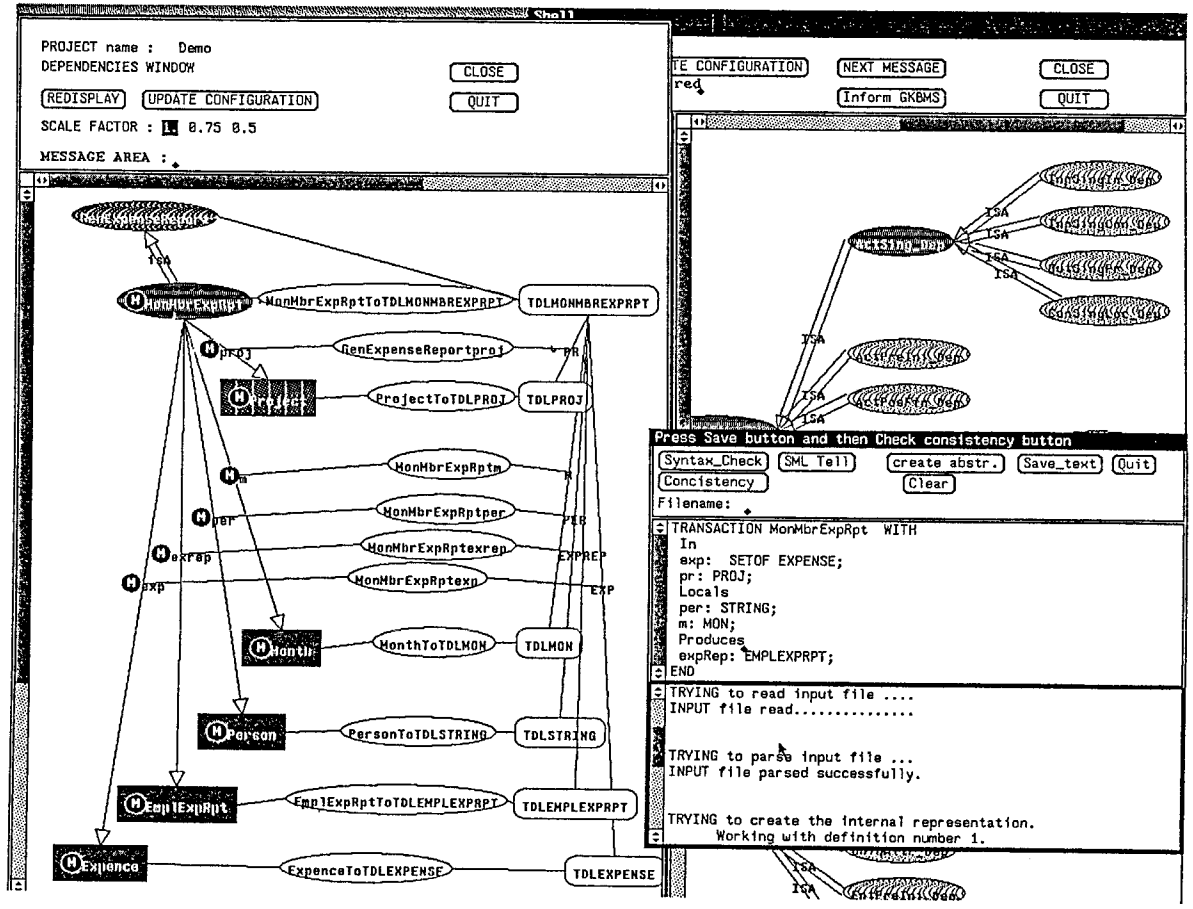

Figure 3.9 Dependency Graph from the Mapping of &GenExpenseReport

4 Mapping Issues

The previous sections outlined a mapping framework which guides the user to synergistically develop a requirements specification and a corresponding design. Guidance is determined by the dependency types supported by the mapping assistant and by our adopted refinement methodology. This mapping framework underconstrains the user in his choice of a design for a given requirements specification. This section points out some of the areas where the designer needs to make hard decisions and discusses some of the alternatives he needs to consider.

4.1 Mapping of Time

Time in Entity Classes: Entity classes that are part of the system model in the requirements specification are generally mapped into design data classes. However, the former are modelled through a declarative description of the allowable histories of their instances, while the latter need to be defined procedurally in terms of state invariants for their instances and operations that create, destroy and update them. The first question that needs an answer in generating a design for such entity classes is: *what kind of information, including historical information, should be kept in each possible state of the conceptual design model?* What historical information needs to be maintained depends, of course, on the queries that might be asked. For example, the following query requires the maintenance of all instances, past and present, of the entity class `TechnicalMeetings`.

```
Which technical meetings pertain to the project daida?
```

As indicated earlier, Telos provides for the representation of both history time (i.e., the history of the application) and belief time (i.e., the history of the knowledge base) through the association of two intervals to every Telos proposition.

To represent temporal information in the design, we need first define in TDL the entity class `TimeInterval` whose instances are the time intervals used in the system:

```
ENTITY CLASS TimeInterval WITH
   UNCHANGING
            from: Date;
   CHANGING
            to: Date;
   UNIQUE
            id(from, to)
END
```

Two useful concepts for conceptual design of databases with temporal information can be adopted from temporal databases [Snodgrass86], [Snodgrass87]: *validity time or existence period* which represents the time when the stored information was true (history time, in our terminology) and *transaction time* which represents the time the information was inserted in the database (belief time). Accordingly, every (design) data class may have two new temporal attributes, `validityTime` and `transactionTime`:

```
ENTITY CLASS Employee ISA Person WITH
   UNCHANGING
      eName: String;
   CHANGING
      eAddress: Address;
      worksOn: SET OF Project;
      salary: Amount;
      validityTime: TimeInterval;
      transactionTime: TimeInterval;
   UNIQUE
      id: (eName);
END
```

For requirements entity classes with invariant extensions, the value of `validityTime` is `AllTime`. Moreover, this attribute needs to be classified as unchanging. Likewise, `transactionTime` might be declared as unchanging and its value might be set to ∞ or to the time system operation was launched.

```
AGGREGATE CLASS Month WITH
   UNCHANGING
      month: 1..12;
      validityTime: [AllTime];
      transactionTime: [SystemLaunchTime];
   INVARIANT
      : (ALL x IN Month)(SOME y IN Date)
        (x.transactionTime = [y..31/12/2000]);
        {* System must be launched between y and
                                    31/12/2000 *}
END
```

Some objects represented in an information system will be inactive in the sense that their validity time has passed (consider, for example, a project that has terminated). Such objects may still be needed, depending on expected usage of the information handled by the system under development. *Relevance period* is another useful concept, intended to provide precisely this information. If the relevance period of an object is greater than its validity time, then past information must be maintained for that object as long as it is its relevance period. One way to structure inactive objects is through "conceptual archives": inactive objects are classified according to the day, week, month or year of their validity time, in addition to their other classifications. Alternatively, inactive objects might be classified according to their relation to the present. For instance, inactive instances of the class `Meeting` may be partitioned into one year collections through classes `LastYearMeetings`, `TwoYearAgoMeetings`, etc. Obviously, this scheme requires update operations. Note that in the DAIDA framework these issues are only dealt with at the design level and will lead to design objects, including data, transaction and script classes, which don't have requirements specification counterparts.

Time in Activity Classes: The declarative representation of temporal constraints in activities needs to be transformed usually into a more fine-grained state-transition view

in the conceptual design. Since TDL treats transactions as atomic database operations, the transcations can only be used as design objects for activities with uninterruptible subactivities. Interruptible subactivities include communication, either between activities or between the system and end users. As with inactive entities, information about activities may have to be maintained after their termination.

Since scripts are long-term events, one may want to store in the information system not only an inactive script's validity time, but also the history of state and transition activations/de-activations. Transitions can be treated as generalized transactions for which the system maintains the input which triggered the transition, the goals (output) performed by the transition and the transition time interval.

As in the case of data classes the notion of relevance period must be applied here to prevent the rapid growth of the amount of past information.

4.2 Mapping of Assertions

In general, a temporal assertion in the requirements specification will have to be analysed and possibly reconfigured in order to be integrated in the system design. Here are some basic alternatives:

From Telos Assertions to TDL Assertions: One of the many difficulties here is that each entity in the constraints of the system model has a time interval associated with it, whereas each data class in the assertions of the conceptual design model is associated with a *validity time* interval which is defined in terms of two time points. For instance, the following constraint of the system model requires that *all expenses of an employee should occur during a meeting of a project in which he is involved*:

```
(Forall x/&Expense)(Exists mt/&Meeting)
        (x.meet = mt) and (x during mt) and
        (mt.proj subsetOf x.participant.proj)
```

This constraint might be mapped into

```
(ALL x IN Expense)(SOME mt IN Meeting)
    (x.meet = mt) AND
    AFTER(x.validityTime.from, mt.validityTime.from) AND
    BEFORE(x.validityTime.to, mt.validityTime.to) AND
    (mt.proj SUBSETOF x.participant.proj)
```

in TDL. Note that the resulting formula is considerably more opaque, due to the mapping of intervals into time points.

From Telos Assertions to Satisfaction by Design: Rather than transforming a constraint in the system model into assertions in the design, the developer may choose to enforce the constraints through system operations and interactions with its environment. Consider, for example,

```
(Forall x/&GenExpenseReport)
    (x  meets this.beginTime + 6mo)  or
    (Exists y/GenExpenseReport) (x  meets y + 6mo)
```

which requires, roughly speaking, that expense reports be generated every six months. Naively, this constraint might be enforced by a script transition that fires every six months and generates a report. However, having the system generate a report is only useful when the system has full information on the subject matter. Accordingly, a more pragmatic enforcement of the constraint may involve a cooperative scheme between the system and users where the system reminds them about an impending deadline, gets the information it needs[7] and proceeds to generate the report. Moreover, the designer may want to add constraints at the requirements or design level to facilitate this information-gathering process for the system. For instance, he may add the constraint that *no expense receipt should be submitted more than a month after the expense actually occurred* (requirements level) or *no expense summary can be inserted in the system more than a month after the relevant meeting took place* (design level). Alternatively, the system may be designed to send out periodic reminders, keep track of meetings – past, present and forthcoming – and send out specific reminders or use some other scheme.

Note that in all of the above scenarios, requirements constraints are enforced by the structure of transactions or scripts and even by the (helpful) behaviour of the environment.

4.3 Mapping of Generalization and Classification Hierarchies

Modelling the world is generally considered a more difficult task than designing a system. Telos recognizes this by offering levels of metaclasses intended to help the user define the concepts that are most appropriate for the modelling task at hand. TDL, on the other hand, is built around a *fixed* set of concepts for conceptual system design and has no use for metaclasses. To deal with this difference, all information associated with a requirements specification beyond the simple class level needs to be collapsed down to simple classes. This collapsing process can be complicated by the presence of generalization hierarchies at metaclass levels.

When dealing with a generalization hierarchy at the metaclass or higher levels, we have to suppress the hierarchy into some metaclass definitions and then have to map the resulting classification hierarchy. In the first step, a class B inherits all attributes from a class A (at a higher level in the generalization hierarchy) and becomes class B'. We may keep class A as is, or we may remove it if we know that it will never be instantiated. In the second step, corresponding classes at a lower level inherit attributes from B', thus eliminating the generalization. We eventually have only classification hierarchies that need to be mapped to TDL.

Elimination of classification hierarchies implies the loss of the ability to talk about sets of classes. Therefore, we have to explicitly associate semantics with them. The methodology employed allows to suppress classification hierarchies by explicitly defining all the knowledge, included in the description of a metaclass, in the instances of the particular metaclass.

[7]Obviously, this is still an idealization of what happens within an organization

5 Current Status and Conclusions

This paper proposes a novel framework for the generation of information system designs from given requirements specifications. The framework is based on a number of premises. Firstly, it adopts the DAIDA architecture which relegates different classes of decisions to different stages of the software development process. Secondly, it assigns a particular role to the mapping assistant in the software development process which involves primarily constraint enforcement and suggestion of basic alternatives. Thirdly, it employs a knowledge engineering approach for the mapping assistant, whereby the relevant knowledge sources are identified and their role in the performance of the mapping task is formally characterized and embedded in the framework's control regime. Finally, a single knowledge representation language is used as an appropriate linguistic vehicle for capturing all types of knowledge relevant to the mapping task.

The implementation of IRIS described in this paper is part of an effort to develop a prototype of the DAIDA environment. This prototype has already been demonstrated and is currently being extended and refined. It runs on SUN workstations and is implemented on top of BIM-Prolog, where the windows and graphics interface have been implemented in SunView and Pixrect.

The prototype implementation of the mapping assistant only employs approximately 30 dependency types at this point. We estimate that a reasonably complete implementation will require 50 -100 such types, in addition to analogous sets of decomposition, satisficing and refinement methods.

Some of the limitations of the implementation are due to the status of the Telos and TDL implementations. In particular, in the current implementation of Telos, performance degrades quite rapidly with the size of the knowledge base. The implementation of TDL supports the insertion, retrieval and update for data and transaction classes. Moreover, most of the assertion language has been implemented, while scripts and functions have only been partly implemented. The Telos and TDL implementations consist of 3 and 1.5Mb of Prolog code respectively and require more than 35Mb of virtual space to run efficiently.

Future plans for the implementation of IRIS include adding more dependency types and decomposition, satisficing and refinement methods. Also, extending and improving the implementations of Telos and TDL. Finally, refining the control structure which utilizes these knowledge sources to guide the generation and the justification of a design.

Bibliography

[Allen81] James F. Allen, *A General Model of Action and Time*, Proceedings 7th IJCAI, Vancouver, BC, Canada, 1981.

[Barron82] John Barron, *Dialogue and Process Design for Interactive Information Systems Using Taxis,* In Proceedings SIGOA Conference on Office Information Systems, Philadelphia, PA, SIGOA Newsletter, Vol. 3, Nos 1 and 2, pp. 12-20, 21-23 June 1982.

[**Borgida85**] A. Borgida, *Features of Languages for the Development of Information Systems at the Conceptual Level,* IEEE Software, Vol. 2, No. 1, Jan. 1985, pp. 63-72.

[**Borgida87**] Alex Borgida, John Mylopoulos, Joachim W. Schmidt and Eric Meirlaen, *Final Version of TDL Design*, Esprit Project DAIDA (892), deliverable DES1.2, Sept. 1987.

[**Borgida89**] Alex Borgida, Matthias Jarke, John Mylopoulos, Joachim W. Schmidt and Yannis Vassiliou, *The Software Development Environment as a Knowledge Base Management System.* in J. W. Schmidt and C. Thanos (Editors), Foundations of Knowledge Base Management. Springer-Verlag, 1989.

[**Chung84**] Lawrence Chung, *An Extended Taxis Compiler,* M.Sc. thesis, Dept. of Computer Science, University of Toronto, Jan. 1984. Also CSRG Technical Note 37, 1984.

[**Chung89**] Lawrence Chung, Panagiotis Katalagarianos, Manolis Marakakis, Michalis Mertikas, John Mylopoulos and Yannis Vassiliou, *From Information System Requirements to Designs: A Mapping Framework*, Technical Report FORTH/CSI/TR/1989/020. Institute of Computer Science - FORTH, Heraklion, November 1989.

[**Findler79**] Findler, N. (editor), *Associative Networks,* Academic Press, 1979.

[**Greenspan84**] S. Greenspan, *Requirements Modelling: The Use of Knowledge Representation Techniques for Requirements Specification,* Ph. D. thesis, Dept. of Computer Science, University of Toronto, 1984.

[**Hayes87**] I. Hayes (editor), *Specification Case Studies*, Prentice Hall International, Englewood Cliffs NJ, 1987.

[**Hull87**] R. Hull and R. King, *Semantic Database Modelling: Survey, Applications and Research Issues*, ACM Computing Reviews 19, No. 3, Sept. 1987.

[**Jackson83**] Michael Jackson, *System Development*, Prentice-Hall, 1983.

[**Jarke86**] M. Jarke (ed), *Development of Advanced Interactive Data-Intensive Applications (DAIDA), Global Design Report,* Esprit-Project 892, Sept. 1986.

[**Jarke89**] Matthias Jarke, Manfred Jeusfeld, Tomas Rose, *A Software Process Data Model for Knowledge Engineering in Information Systems.* Information Systems, Vol.14, No.3, Fall 1989.

[**Katalagarianos89**] Panos Katalagarianos, Manolis Marakakis, Michalis Mertikas, Yannis Vassiliou, *CML/Telos - TDL Mapping Assistant: Architecture and Development,* Esprit Project 892 (DAIDA), del. DES2.3, Institute of Computer Science, Foundation for Research and Technology, Heraklion, Crete, Greece, Febr. 1989.

[**Koubarakis89**] M. Koubarakis, J. Mylopoulos, M. Stanley and A. Borgida, *Telos: Features and Formalization,* Technical Report KRR-TR-89-4, Dept. of Computer Science, Univ. of Toronto, 1989.

[Mylopoulos86] *The Role of Knowledge Representation in the Development of Specifications,* In H. J. Kugler (ed.): Information Processing, Elsevier Science Publishers B. V., North-Holland, 1986.

[Roman85] Gruia-Catalin Roman, *A Taxonomy of Current Issues in Requirements Engineering,* In IEEE Computer, pp. 14-21, Apr., 1985.

[Schmidt88] J. Schmidt, H. Eckhardt, and F. Matthes, *DBPL Report.* DBPL-Memo 111-88, Fachbereich Informatik, johann Wolfgang Goethe-Universitat, Frankfurt, West Germany, 1988.

[Snodgrass86] Richard Snodgrass, *Temporal Databases,* Computer, September 1986, pp. 35-42.

[Snodgrass87] Richard Snodgrass, *The Temporal Query Language TQuel,* In ACM Transactions on Database Systems, 1987.

[Stanley86] M. Stanley, *A Formal Semantics for CML,* M. Sc. thesis, Dept. of Computer Science, University of Toronto, 1986.

[Zave81] Pamela Zave and Raymond T. Yeh, *Executable Requirements for Embedded Systems,* In Proceedings fifth International Conf. on Software Engineering, pp. 295-304, 1981.